Contents

1 Hoare Logic

In the most straightforward sense, hoare logic is a system or framework for deducing valid correctness of formulas in a mechanical way, using a set of derivation rules. It enables reasoning about the correctness about a program's syntax, without having to consider the operational semantics.

Hoare logic distinguishes itself from operational semantics in that it's intended to represent an idealized verifier as opposed to an idealized interpreter.

1.1 Hoare Triples

The basic unit of reasoning or *judgement* in hoare logic is the *hoare triple*, which has the following form:

$$\{P\}\ s\ \{Q\}$$

where

- P is the *precondition*, a logical assertion (i.e. formula) that must hold before the execution of statement s.
- S is a statement or sequence of statements in the programming language being analyzed
- Q is the *postcondition*, a logical assertion that must hold after the execution of statement S, provided that P held before execution.

Hoare triples represent a form of partial correctness, in that the program is correct if it terminates; otherwise the program can behave arbitrarily. We intuitively want a hoare triple to be valid as this represents a correct program, some basic examples.

$$\{\texttt{true}\} \quad \texttt{b} := 4 \quad \{\texttt{b} = 4\} \tag{1}$$

$$\{\texttt{a} = 2\} \quad \texttt{b} := 2 * \texttt{a} \quad \{\texttt{a} = 2 \vee \texttt{b} = 4\} \tag{2}$$

$$\{\texttt{b} \geqslant 5\} \quad \texttt{b} := \texttt{b} + 1 \quad \{\texttt{b} \geqslant 6\} \tag{3}$$

$$\{\texttt{false}\} \quad \texttt{skip} \quad \{\texttt{b} = 10\} \tag{4}$$

$$\{\texttt{true}\} \quad \texttt{while i} \neq 10 \texttt{ do i} := \texttt{i} + 1 \quad \{\texttt{i} = 10\} \tag{5}$$

We can see that the first 3 hoare triples are quite trivially valid. The 4th one is considered *vacuously true* as the precondition can never be satisfied, and thus the program can behave arbitrarily. To elaborate we consider the 4th triple equivalent to the proposition

$$\texttt{false} \rightarrow (\texttt{b} = 10)$$

from propositional logic we know that any proposition of the form $F \rightarrow P$ is true if F is false, regardless of the truth value of P.

If you're still confused remember that

$$P \rightarrow Q \equiv \neg P \vee Q \equiv \text{whenever P is true, Q must be true}$$

thus if P is false, $P \rightarrow Q$ is true regardless of Q. If we put this as an analogy consider the following

If it rains tomorrow, I will bring an umbrella.

if it doesn't rain then I haven't broken my promise, the only way I break my promise is if the *antecedent* "it rains tomorrow" is true and the *consequent* "I will bring an umbrella" is false. Thus if the condition P never holds, we consider the proposition to be vacuously true.

For the 5th triple we have no guarantee that the loop will terminate (i.e. if i starts at 11), but if the loop does terminate we must have the case that $\texttt{i} = 10$, hence this is our postcondition. Note that the 5th triple is a good example of partial correctness, in that the hoare triple's validity does not guarantee termination of the program. It's a conditional assertion that *if* the program terminates, *then* the postcondition must hold.

1.2 Validity of Hoare Triples

We denote the validity of a hoare triple as

$$\models \{P\}\ s\ \{Q\}$$

If we denote a *state* as $\sigma : \texttt{Vars} \to \mathbb{Z}$ then we can define partial validity of a hoare triple as follows:

$$\sigma \models P \land \exists \sigma.\ \langle S, \sigma\rangle \to^* \sigma' \to \sigma' \models Q$$

In plain English, this means that if the precondition P holds in some state σ, and if executing statement S from state σ eventually terminates in some state σ', then the postcondition Q must hold in state σ'.

$\langle s, \sigma\rangle \to^* \sigma'$ denotes that the statement S evaluates to a state σ' after a finite number of steps, starting from state σ. Sometimes this is also written as

$$\langle s, \sigma\rangle \Downarrow \sigma'$$

1.2.1 Total Correctness

We denote a hoare triple exhibiting total correctness as

$$[P]\ s\ [Q]$$

We can define total correctness as follows:

$$\sigma \models P \to \underbrace{\exists \sigma'.\ \langle S, \sigma\rangle \to^* \sigma'}_{\text{termination condition}} \land \sigma' \models Q$$

The main distinguishing factor being that the termination condition is now a necessary condition, i.e. its part of the consequent of the implication as opposed to the antecedent meaning that it's no longer a given that we assume termination.
Again in plain English the total correctness of a hoare triple means that if the precondition P holds in some state σ, then executing statement S from state σ must eventually terminate in some state σ', and the postcondition Q must hold in state σ'.

1.3 Hoare Rules

Hoare logic provides a set of derivation rules that can be used to deduce the validity of hoare triples. These rules are typically presented in the form of inference rules, which allow us to derive new hoare triples from existing ones. The general syntax which hoare rules follow is as follows

$$\frac{\vdash \{P_1\}\ s_1\ \{Q_1\} \qquad \dots \qquad \vdash \{P_n\}\ s_n\ \{Q_n\}}{\vdash \{P\}\ s\ \{Q\}}\ \textsc{Rule}$$

For brevity we omit the $\vdash$ symbol in the following rules, as it's implied that we're deriving valid hoare triples.

In english we can read the general inference rule as if we have valid hoare triples $\{P_1\}\ s_1\ \{Q_1\}, \dots, \{P_n\}\ s_n\ \{Q_n\}$ then we can derive the valid hoare triple $\{P\}\ s\ \{Q\}$. In other words if the premise of hoare triples are provable / valid then the conclusion is also provable / valid.

A set of common hoare rules are as follows:

$$\frac{}{\{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assign} \qquad \frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1;\ s_2\ \{R\}}\ \textsc{Seq (Composition)}$$

$$\frac{\{P \land B\}\ s_1\ \{Q\} \qquad \{P \land \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}}\ \textsc{If} \qquad \frac{\{I \land B\}\ s\ \{I\} \qquad P \to I \qquad I \land \neg B \to Q}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{Q\}}\ \textsc{While}$$

$$\frac{P' \to P \qquad \{P\}\ s\ \{Q\} \qquad Q \to Q'}{\{P'\}\ s\ \{Q'\}}\ \textsc{Conseq}$$

1.3.1 Pre-and Postcondition Rule (Consequence Rule)

The pre-and postcondition rule can actually be broken up into two separate rules

$$\frac{P' \to P \qquad \{P\}\ s\ \{Q\}}{\{P'\}\ s\ \{Q\}}\ \textsc{Precond. strengthening} \qquad\qquad \frac{\{P\}\ s\ \{Q\} \qquad Q \to Q'}{\{P\}\ s\ \{Q'\}}\ \textsc{Postcond. weakening}$$

The term *strengthening* refers to making the precondition more strict, i.e. making it harder to satisfy by having a stronger condition or demand on the initial state. Conversely *weakening* refers to making the postcondition less strict, i.e. making it easier to satisfy by having a weaker condition or demand on the final state.

For example strengthening a condition would be something like

$$x > 0 \xrightarrow{\text{stricter}} x > 5$$

whereas weakening a condition would be something like

$$x = 1 \xrightarrow{\text{weaker}} x \geqslant 1$$

Hence the precondition strengthening rule states that if we have a valid hoare triple $\{P\}\ s\ \{Q\}$ and we can prove that P' is stronger than P (i.e. P' implies P) then we can derive the valid hoare triple $\{P'\}\ s\ \{Q\}$. Similarly the postcondition weakening rule states that if we have a valid hoare triple $\{P\}\ s\ \{Q\}$ and we can prove that Q' is weaker than Q (i.e. Q implies Q') then we can derive the valid hoare triple $\{P\}\ s\ \{Q'\}$.

1.3.2 While Rule

Central to reasoning about loops is the concept of a loop invariant - an invariant being a property that holds true before and after each iteration of the loop. In the while rule we denote the loop invariant as I with the following conditions:

- Base case - $P \to I$ - the loop invariant must hold before the first iteration of the loop, i.e. the precondition must imply the invariant.
- Inductive step - $\{I \wedge B\}\ s\ \{I\}$ - the loop invariant must be preserved by the loop body, i.e. if the invariant holds before an iteration of the loop and the loop condition B is true, then after executing the loop body s the invariant must still hold.
- Termination - $I \wedge \neg B \to Q$ - when the loop terminates (i.e. the loop condition B is false), the invariant combined with the negation of the loop condition must imply the postcondition.

Formally we express this via the while rule as shown above. Often times a more common specification of the while loop is one where we fix I to some P and hence we have the following simplified while rule:

$$\frac{\{P \wedge B\}\ s\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{P \wedge \neg B\}}\ \textsc{While (simple)}$$

In other words for the simplified rule we are tying the loop invariant to the precondition, and the postcondition is simply the invariant combined with the negation of the loop condition.

1.3.3 Arrays

Using the theory of arrays we can define the corresponding hoare rule for an array assignment

$$\frac{}{\{Q[a\langle e_1 \lhd e_2\rangle := a]\}\ (a[e_1] := e_2)\ \{Q\}}\ \textsc{Array Assign}$$

The precondition here is derived from the postcondition by substituting all occurrences of the array a with the updated array $a\langle e_1 \lhd e_2\rangle$, which represents the array a after updating index e_1 with value e_2.

1.3.4 Application of Hoare Rules

- Assignment - To prove that Q holds *after* the assignment $x := E$, it is sufficient to show that $Q[E/x]$ holds *before* the assignment. Let's consider the following example:

$$\{y = 4\}\, x := 4\, \{y = x\} \tag{6}$$
$$\{x + 1 = n\}\, x := x + 1\, \{x = n\} \tag{7}$$
$$\{y = x\}\, y := 2\, \{y = x\} \tag{8}$$
$$\{z = 3\}\, y := x\, \{z = 3\} \tag{9}$$

  1. We want to prove that $y = x$ holds after the assignment $x := 4$. By the assignment rule we need to show that $y = 4$ holds before the assignment, which is given in the precondition. Thus this hoare triple is valid.
  2. We want to prove that $x = n$ holds after the assignment $x := x + 1$. By the assignment rule we need to show that $x + 1 = n$ holds before the assignment, which is given in the precondition. Thus this hoare triple is valid.
  3. We want to prove that $y = x$ holds after the assignment $y := 2$. By the assignment rule we need to show that $y = x[2/y]$ holds before the assignment, i.e. $2 = x$. This is not given in the precondition, thus this hoare triple is not valid.
  4. We want to prove that $z = 3$ holds after the assignment $y := x$. By the assignment rule we need to show that $z = 3$ holds before the assignment, which is given in the precondition. Thus this hoare triple is valid.

- Precond. strengthening - To prove that P' holds *before* the statement s, it is sufficient to show that P' implies P and that P holds *before* the statement s.

- Postcond. weakening - To prove that Q' holds *after* the statement s, it is sufficient to show that Q implies Q' and that Q holds *after* the statement s. Let's consider that we can prove the following

$$\{\top\}\, s\, \{x = y \ \wedge\ z = 2\}$$

  applying postcondition weakening Let's examine which of the following hoare triples are valid:

$$\{\top\}\, s\, \{x = y\} \tag{1}$$
$$\{\top\}\, s\, \{z = 2\} \tag{2}$$
$$\{\top\}\, s\, \{z > 0\} \tag{3}$$
$$\{\top\}\, s\, \{\forall y.\ x = y\} \tag{4}$$
$$\{\top\}\, s\, \{\exists y.\ x = y\} \tag{5}$$

  1. We want to prove that $x = y$ holds after the statement s. By the postcondition weakening rule we need to show that $x = y \ \wedge\ z = 2 \rightarrow x = y$, which is true. Thus this hoare triple is valid.
  2. We want to prove that $z = 2$ holds after the statement s. By the postcondition weakening rule we need to show that $x = y \ \wedge\ z = 2 \rightarrow z = 2$, which is true. Thus this hoare triple is valid.
  3. We want to prove that $z > 0$ holds after the statement s. By the postcondition weakening rule we need to show that $x = y \ \wedge\ z = 2 \rightarrow z > 0$, which is true. Thus this hoare triple is valid.
  4. We want to prove that $\forall y.\ x = y$ holds after the statement s. By the postcondition weakening rule we need to show that $x = y \ \wedge\ z = 2 \rightarrow \forall y.\ x = y$, which is false (e.g. if $x = 1$ and $y = 2$). Thus this hoare triple is not valid.
  5. We want to prove that $\exists y.\ x = y$ holds after the statement s. By the postcondition weakening rule we need to show that $x = y \ \wedge\ z = 2 \rightarrow \exists y.\ x = y$, which is true (e.g. if $x = 1$ then $\exists y.\ x = y$ is satisfied by $y = 1$). Thus this hoare triple is valid.

- Composition - To prove that R holds *after* the sequence of statements $s_1;\ s_2$, it is sufficient to show that there exists some intermediate assertion Q such that Q holds *after* statement s_1 and R holds *after* statement s_2. As an example Let's consider

$$\{\top\}\, x := 2;\ y := x\, \{y = 2 \wedge x = 2\}$$

  We can prove this as follows:

$$\dfrac{\dfrac{}{\{\top\}\, x := 2\, \{x = 2\}}\ \textsc{Assign} \qquad \dfrac{}{\{x = 2\}\, y := x\, \{y = 2 \wedge x = 2\}}\ \textsc{Assign}}{\{\top\}\, x := 2;\ y := x\, \{y = 2 \wedge x = 2\}}\ \textsc{Seq}$$

- If - To prove that Q holds *after* the conditional statement `if B then` s_1 `else` s_2, it is sufficient to show that Q holds *after* statement s_1 when B is true, and that Q holds *after* statement s_2 when B is false. As an example Let's consider

$$\{\top\}\ \texttt{if}\ x > 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y \geqslant 0\}$$

We can prove this as follows:

$$\frac{\dfrac{}{\{x > 0\}\, y := x\, \{y \geqslant 0\}}\ \textsc{Assign} \qquad \dfrac{}{\{x \leqslant 0\}\, y := -x\, \{y \geqslant 0\}}\ \textsc{Assign}}{\{\top\}\ \texttt{if}\ x > 0\ \texttt{then}\ y := x\ \texttt{else}\ y := -x\ \{y \geqslant 0\}}\ \textsc{If}$$

Here we used the fact that $\top \wedge (x > 0) \to x > 0$ and $\top \wedge \neg(x > 0) \to x \leqslant 0$. So we can also write this as

$$\frac{\{(P := \top) \wedge (B := (x > 0))\}\, y := x\, \{y \geqslant 0\} \qquad \{P \wedge \neg B\}\, y := -x\, \{y \geqslant 0\}}{\{\top\}\ \texttt{if}\ B\ \texttt{then}\ \underbrace{y := x}_{s_1}\ \texttt{else}\ \underbrace{y := -x}_{s_2}\ \{y \geqslant 0\}}\ \textsc{If}$$

> The intuition here is that we're splitting the reasoning into two separate cases, one for when the condition B is true and one for when it's false. In each case we can use the appropriate precondition to reason about the corresponding branch of the conditional statement.

- While (simple) - To prove that $P \wedge \neg B$ holds *after* the while loop `while B do s`, it is sufficient to show that P holds *before* the loop, and that P is preserved by the loop body s when B is true. As an example Let's consider

```
1 i = 1; j = 1; while i < n do { j = j + i; i = i + 1 }
```

Let's pick the invariant I as

$$I = j \geqslant 1$$

the first question to ask is, does this qualify as a loop invariant?

  - Initialization - does the invariant hold before the first iteration of the loop? Yes, as $j = 1$ initially and thus $j \geqslant 1$ holds.
  - Preservation - preservation requires that we must prove

$$\{j \geqslant 1 \wedge i \leqslant n\}\, (j := j + i;\ i := i + 1)\, \{j \geqslant 1\}$$

The central requirement of this is that we have

$$j + i \geqslant 1 \text{ given } j \geqslant 1 \wedge i \leqslant n$$

which is *not derivable* as i could for example be very negative in the precondition.

> For example Let's take
>
> $$j = 1, i = -100, n = -99$$
>
> Then we have
>
> $$j \geqslant 1 \to 1 \geqslant 1 \qquad \texttt{(true)}$$
> $$i \leqslant n \to -100 \leqslant -99 \qquad \texttt{(true)}$$
> $$j + i \geqslant 1 \to 1 + (-100) \geqslant 1 \to -99 \geqslant 1 \qquad \texttt{(false)}$$

Thus this is not a valid loop invariant. The crux of the issue being that I is not an *inductive invariant*, i.e. it does not hold before and after each iteration of the loop.

Let's consider the following strengthening

$$I = j \geqslant 1 \wedge i \geqslant 1$$

Now we have

- Initialization - does the invariant hold before the first iteration of the loop? Yes, as $j = 1$ and $i = 1$ initially and thus $j \geqslant 1 \land i \geqslant 1$ holds.
- Preservation - we must prove

$$\{j \geqslant 1 \land i \geqslant 1 \land i \leqslant n\}\ (j := j + i;\ i := i + 1)\ \{j \geqslant 1 \land i \geqslant 1\}$$

$$\begin{aligned} j' = j + i \geqslant 1 + 1 \geqslant 1 && \text{(after } \texttt{j = j + i}\text{)} \\ i' = i + 1 \geqslant 1 + 1 \geqslant 1 && \text{(after } \texttt{i = i + 1}\text{)} \\ \therefore j' \geqslant 1 \land i' \geqslant 1 \land \neg(i \leqslant n) && \text{(after loop)} \end{aligned}$$

Thus this is a valid loop invariant.

- Array Assignment - To prove that Q holds *after* the array assignment $a[e_1] := e_2$, it is sufficient to show that $Q[a\langle e_1 \triangleleft e_2\rangle := a]$ holds *before* the assignment. Let's consider the following example:

$$\{i = 1\}\ a[i] := 3;\ a[1] := 2\ \{a[i] = 3\}$$

We can attempt to prove this as follows:

$$\dfrac{\dfrac{}{\{P := i = 1\}\ a[i] := 3\ \{Q := a[i] = 3\}}\ \textsc{Array Assign} \qquad \dfrac{}{\{Q\}\ a[1] := 2\ \{R := a[i] = 2\}}\ \textsc{Array Assign}}{\{P\}\ a[i] := 3;\ a[1] := 2\ \{R\}}\ \textsc{Seq}$$

Hence our example is not valid. Let's consider another example

```
// @pre i = 0 ∧ n > 0
// @post ∀j. (0 ⩽ j < n) → a[j] = 0
while i < n do {
    a[i] = 0;
    i = i + 1;
}
```

If we now wanted to define an invariant I we could pick it as follows

- Initialization - Our invariant must hold before the first iteration of the loop, i.e. $P \to I$. We can pick

$$I := \forall j.\ (0 \leqslant j < i) \to a[j] = 0$$

- Preservation - We must prove that

$$\{I \land B\}\ (a[i] := 0;\ i := i + 1)\ \{I\}$$

where B is the loop condition $i < n$. We can prove this as follows:

$$\begin{aligned} I &\to \forall j.\ (0 \leqslant j < i) \to a[j] = 0 && \text{(before loop body)} \\ B &\to i < n && \text{(before loop body)} \\ a[i] := 0 &\to a[i] = 0 && \text{(after } a[i] := 0\text{)} \\ i := i + 1 &\to i' = i + 1 && \text{(after } i := i + 1\text{)} \\ I' &\to \forall j.\ (0 \leqslant j < i') \to a[j] = 0 && \text{(after loop body)} \\ &\equiv \forall j.\ (0 \leqslant j < i + 1) \to a[j] = 0 \\ &\equiv \forall j.\ ((0 \leqslant j < i) \lor (j = i)) \to a[j] = 0 \\ &\equiv \forall j.\ ((0 \leqslant j < i) \to a[j] = 0) \land ((j = i) \to a[j] = 0) \\ &\equiv (\forall j.\ (0 \leqslant j < i) \to a[j] = 0) \land (\forall j.\ (j = i) \to a[i] = 0) \\ &\equiv (\forall j.\ (0 \leqslant j < i) \to a[j] = 0) \land (a[i] = 0) \\ &\equiv I \land (a[i] = 0) && \text{(after loop body)} \end{aligned}$$

The preservation proof uses some logical equivalences which are worth noting:

- *Distributive Law*
$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$
- *Universal Quantifier Distribution*
$$\forall x.\ (P(x) \wedge Q(x)) \equiv (\forall x.\ P(x)) \wedge (\forall x.\ Q(x))$$
- *Implication Distribution*
$$(P \vee Q) \to R \equiv (P \to R) \wedge (Q \to R)$$
- *Point simplification*
$$\forall x.\ (x = c) \to P(x) \equiv P(c)$$

In addition there is the successor split notion

$$\{0, \ldots, i\} \equiv \{0, \ldots, i-1\} \cup \{i\}$$

Hence we were able to say
$$(0 \leqslant j < i+1) \equiv (0 \leqslant j < i) \vee (j = i)$$

1.4 Soundness and Completeness of Hoare Logic

As a reminder hoare triples can be either valid or derivable. A hoare triple $\{P\}\ s\ \{Q\}$ is said to be *valid* if it holds true in the semantics of the programming language, i.e. if executing statement s from any state satisfying precondition P will always terminate in a state satisfying postcondition Q. A hoare triple is said to be *derivable* if it can be proven using the rules of hoare logic.

So the intuition here is that validity is a semantic notion, i.e. it relates to the actual behavior of programs, whereas derivability is a syntactic notion, i.e. it relates to the formal system of hoare logic itself.

- Soundness - If a hoare triple $\{P\}\ s\ \{Q\}$ is derivable using the hoare rules, then it is valid. In other words if we can prove a hoare triple using the rules of hoare logic, then it must be true in the semantics of the programming language. Formally
$$\vdash \{P\}\ s\ \{Q\} \ \to \models \{P\}\ s\ \{Q\}$$
- Completeness - If a hoare triple $\{P\}\ s\ \{Q\}$ is valid, then it is derivable using the hoare rules. In other words if a hoare triple is true in the semantics of the programming language, then we can prove it using the rules of hoare logic. Formally
$$\models \{P\}\ s\ \{Q\} \ \to \vdash \{P\}\ s\ \{Q\}$$

1.4.1 Relative Completeness

The rules for precondition strengthening and postcondition weakening require us checking the validity of implications between assertions, i.e. checking whether $P' \to P$ or $Q \to Q'$ hold. This is a non-trivial task and in general is undecidable for arbitrary assertions.
Hence we say that hoare logic is *relatively complete* with respect to the underlying logic of assertions. This means that if we have a complete proof system for the underlying logic of assertions, then we can derive any valid hoare triple using the rules of hoare logic. In other words, the completeness of hoare logic is contingent upon our ability to reason about the assertions themselves.

2 A simple language: Nano

2.1 Syntax

We start by defining a simple imperative programming language called Nano. The syntax of Nano is defined as follows:

$$
\begin{aligned}
\text{Expr} = \mathtt{e} ::= {} & \mathtt{n} \mid \mathtt{x} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{e}_1 - \mathtt{e}_2 \mid \mathtt{e}_1 * \mathtt{e}_2 \\
\text{BExpr} = \mathtt{b} ::= {} & \top \mid \bot \mid \mathtt{e}_1 = \mathtt{e}_2 \mid \mathtt{e}_1 \leqslant \mathtt{e}_2 \mid \neg \mathtt{b} \mid \mathtt{b}_1 \wedge \mathtt{b}_2 \mid \mathtt{b}_1 \vee \mathtt{b}_2 \\
\text{Stmt} = \mathtt{s} ::= {} & \mathtt{skip} && \text{(noop)} \\
& \mid \mathtt{x} := \mathtt{e} && \text{(assignment)} \\
& \mid \mathtt{s}_1 ; \ \mathtt{s}_2 && \text{(sequence)} \\
& \mid \mathtt{if\ b\ then\ s}_1 \ \mathtt{else\ s}_2 && \text{(conditional)} \\
& \mid \mathtt{while\ b\ do\ s} && \text{(loop)}
\end{aligned}
$$

2.2 Operational Semantics

As a reminder, we define the relation $\langle s, \sigma \rangle \to^* \sigma'$ to mean that the statement s evaluates to a state σ' after a finite number of steps, starting from state σ.

2.2.1 Expression Evaluation

$$\frac{}{\langle n, \sigma \rangle \to^* n} \textsc{Num} \qquad \frac{}{\langle x, \sigma \rangle \to^* \sigma(x)} \textsc{Var} \qquad \frac{\langle e_1, \sigma \rangle \to^* n_1 \qquad \langle e_2, \sigma \rangle \to^* n_2}{\langle e_1 + e_2, \sigma \rangle \to^* n_1 + n_2} \textsc{Plus}$$

$$\frac{\langle e_1, \sigma \rangle \to^* n_1 \qquad \langle e_2, \sigma \rangle \to^* n_2}{\langle e_1 - e_2, \sigma \rangle \to^* n_1 - n_2} \textsc{Minus} \qquad \frac{\langle e_1, \sigma \rangle \to^* n_1 \qquad \langle e_2, \sigma \rangle \to^* n_2}{\langle e_1 * e_2, \sigma \rangle \to^* n_1 * n_2} \textsc{Times}$$

2.2.2 Boolean Expression Evaluation

$$\frac{}{\langle \top, \sigma \rangle \to^* \top} \textsc{True} \qquad \frac{}{\langle \bot, \sigma \rangle \to^* \bot} \textsc{False} \qquad \frac{\langle e_1, \sigma \rangle \to^* n_1 \qquad \langle e_2, \sigma \rangle \to^* n_2}{\langle e_1 = e_2, \sigma \rangle \to^* (n_1 = n_2)} \textsc{Eq}$$

$$\frac{\langle e_1, \sigma \rangle \to^* n_1 \qquad \langle e_2, \sigma \rangle \to^* n_2}{\langle e_1 \leqslant e_2, \sigma \rangle \to^* (n_1 \leqslant n_2)} \textsc{Leq} \qquad \frac{\langle b_1, \sigma \rangle \to^* \bot}{\langle b_1 \wedge b_2, \sigma \rangle \to^* \bot} \textsc{And1} \qquad \frac{\langle b_2, \sigma \rangle \to^* \bot}{\langle b_1 \wedge b_2, \sigma \rangle \to^* \bot} \textsc{And2}$$

$$\frac{\langle b_1, \sigma \rangle \to^* v_1 \qquad \langle b_2, \sigma \rangle \to^* v_2}{\langle b_1 \wedge b_2, \sigma \rangle \to^* v_1 \wedge v_2} \textsc{And}$$

Example 2.1. Let's consider the following evaluation:

$$x + 1 \leqslant 3 \quad \sigma[x \mapsto 2]$$

We can evaluate this as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma \rangle \to^* 2} \textsc{Var} \qquad \dfrac{}{\langle 1, \sigma \rangle \to^* 1} \textsc{Num}}{\langle x + 1, \sigma \rangle \to^* 3} \textsc{Plus} \qquad \dfrac{}{\langle 3, \sigma \rangle \to^* 3} \textsc{Num}}{\langle x + 1 \leqslant 3, \sigma \rangle \to^* \top} \textsc{Leq}$$

2.2.3 Statement Evaluation

$$\frac{}{\langle \mathtt{skip}, \sigma \rangle \to^* \sigma} \textsc{Skip} \qquad \frac{\langle e, \sigma \rangle \to^* n}{\langle x := e, \sigma \rangle \to^* \sigma[x \mapsto n]} \textsc{Assign} \qquad \frac{\langle s_1, \sigma \rangle \to^* \sigma' \qquad \langle s_2, \sigma' \rangle \to^* \sigma''}{\langle s_1 ; \ s_2, \sigma \rangle \to^* \sigma''} \textsc{Seq}$$

$$\frac{\langle b, \sigma \rangle \to^* \top \qquad \langle s_1, \sigma \rangle \to^* \sigma'}{\langle \mathtt{if\ b\ then\ s}_1 \ \mathtt{else\ s}_2, \sigma \rangle \to^* \sigma'} \textsc{IfTrue} \qquad \frac{\langle b, \sigma \rangle \to^* \bot \qquad \langle s_2, \sigma \rangle \to^* \sigma'}{\langle \mathtt{if\ b\ then\ s}_1 \ \mathtt{else\ s}_2, \sigma \rangle \to^* \sigma'} \textsc{IfFalse}$$

$$\frac{\langle b, \sigma \rangle \to^* \bot}{\langle \mathtt{while\ b\ do\ s}, \sigma \rangle \to^* \sigma} \textsc{WhileFalse} \qquad \frac{\langle b, \sigma \rangle \to^* \top \qquad \langle s, \sigma \rangle \to^* \sigma' \qquad \langle \mathtt{while\ b\ do\ s}, \sigma' \rangle \to^* \sigma''}{\langle \mathtt{while\ b\ do\ s}, \sigma \rangle \to^* \sigma''} \textsc{WhileTrue}$$

Example 2.2. Simple Assignment
Let's try and evaluate the following statement:

$$\langle x := x + 1, \sigma[x \mapsto 2] \rangle \to^* \sigma'$$

We can evaluate this as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma \rangle \to^* 2}\ \textsc{Var} \qquad \dfrac{}{\langle 1, \sigma \rangle \to^* 1}\ \textsc{Num}}{\langle x + 1, \sigma \rangle \to^* 3}\ \textsc{Plus}}{\langle x := x + 1, \sigma \rangle \to^* \sigma[x \mapsto 3]}\ \textsc{Assign}$$

Example 2.3. If Statement
Let's try and evaluate the following statement:

$$\langle \texttt{if } x + 1 \leqslant 3 \texttt{ then } x := x - 1 \texttt{ else skip}, \sigma[x \mapsto 1] \rangle \to^* \sigma'$$

We can understand this evaluation as consisting of two parts, first we need the condition to evaluate to true, and then we need the assignment to evaluate. We can represent this as follows.

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma \rangle \to^* 2}\ \textsc{Var} \qquad \dfrac{}{\langle 1, \sigma \rangle \to^* 1}\ \textsc{Num}}{\langle x + 1, \sigma \rangle \to^* 3}\ \textsc{Plus} \qquad \dfrac{}{\langle 3, \sigma \rangle \to^* 3}\ \textsc{Num}}{\langle x + 1 \leqslant 3, \sigma \rangle \to^* \top}\ \textsc{Leq (condition)}$$

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma \rangle \to^* 2}\ \textsc{Var} \qquad \dfrac{}{\langle 1, \sigma \rangle \to^* 1}\ \textsc{Num}}{\langle x - 1, \sigma \rangle \to^* 1}\ \textsc{Minus}}{\langle x := x - 1, \sigma \rangle \to^* \sigma[x \mapsto 1]}\ \textsc{Assign (then branch)}$$

$$\dfrac{\langle x + 1 \leqslant 3, \sigma \rangle \to^* \top \qquad \langle x := x - 1, \sigma \rangle \to^* \sigma[x \mapsto 1]}{\langle \texttt{if } x + 1 \leqslant 3 \texttt{ then } x := x - 1 \texttt{ else skip}, \sigma \rangle \to^* \sigma[x \mapsto 1]}\ \textsc{IfTrue}$$

3 Automating Hoare Logic: Part I

An observation which you might have already made is that proof writing can be tedious, hence there is an incentive to automate the process. The general idea is that

1. We are given some loop invariants
2. The program automates the remaining reasoning steps

3.1 Verification Conditions (VCs)

Central to the automation of hoare logic is the concept of *verification conditions*. A verification condition is a logical formula ϕ such that if ϕ is valid, then the corresponding hoare triple $\{P\}\ s\ \{Q\}$ is also valid, i.e. our program is correct.
Deductive verification generally has two main components

- Generation of VCs - This involves generating the verification conditions from the program and its specifications (preconditions, postconditions, and loop invariants). This is typically done using the rules of hoare logic.
- Discharging VCs - This involves proving the validity of the generated verification conditions. This is typically done using automated theorem provers or SMT solvers.

3.1.1 Weakest Precondition (WP) Calculus

Let's start with an idea, say we want to verify

$$\{P\}\ s\ \{Q\}$$

We start from Q and working backwards compute a formula

$$\mathtt{wp}(s, Q) \equiv \textit{ weakest precondition } (s) \text{ w.r.t } (Q)$$

such that if $\mathtt{wp}(s, Q)$ holds before executing s, then Q is guaranteed to hold after executing s. Thus

$$\models \{P\}\ s\ \{Q\} \iff P \to \mathtt{wp}(s, Q) \tag{10}$$

Equivalently we can express this as follows

$$(s \models \mathtt{wp}(s, Q) \land s \to^* s') \to s' \models Q$$

So if a state s satisfies the weakest precondition $\mathtt{wp}(s, Q)$, and s evaluates to s', then s' satisfies Q.

Formally the weakest precondition $\mathtt{wp}(s, Q)$ is a predicate transformer

$$\mathtt{wp}(s, Q) : \mathtt{Stmt} \times \mathtt{FOL} \to \mathtt{FOL}$$

defined inductively on the structure of statements as follows:

- Skip
$$\mathtt{wp}(\mathtt{skip}, Q) = Q$$
- Assignment
$$\mathtt{wp}(x := e, Q) = Q[e/x]$$
where $Q[e/x]$ denotes the substitution of expression e for variable x in formula Q.
- Array Assignment
$$\mathtt{wp}(a[e_1] := e_2, Q) = Q[a\langle e_1 \triangleleft e_2\rangle/a]$$
- Return
$$\mathtt{wp}(\mathtt{return}\ e, Q) = Q[\mathtt{result} \mapsto e]$$
- Sequence
$$\mathtt{wp}(s_1;\ s_2, Q) = \mathtt{wp}(s_1, \mathtt{wp}(s_2, Q))$$
For an arbitrary length sequence of statements we have
$$\mathtt{wp}(s_1;\ \ldots;\ s_n, Q) \Leftrightarrow \mathtt{wp}(s_1;\ \ldots;\ s_{n-1}, \mathtt{wp}(s_n, Q))$$
- If
$$\mathtt{wp}(\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, Q) = (b \to \mathtt{wp}(s_1, Q)) \land (\neg b \to \mathtt{wp}(s_2, Q))$$
- While
$$\mathtt{wp}(\mathtt{while}\ [I]\ b\ \mathtt{do}\ s, Q) = I$$
Importantly this is only sound if we also include the verification conditions for loops which we will discuss shortly.

3.1.2 An intuition for while loops

Loops present a more nuanced case to consider. To provide some intuition for how we construct wp for loops Let's first consider a loop

$$W \triangleq \mathtt{while}\ b\ \mathtt{do}\ s$$

semantically this construction is equivalent to

$$\mathtt{if}\ b\ \mathtt{then}\ (s; W)\ \mathtt{else}\ \mathtt{skip}$$

i.e. the loop execution is conditional on the loop guard b. This allows us to apply the If rule, i.e the weakest precondition we get is

$$\mathtt{wp}(W, Q) = (b \to \mathtt{wp}(s; W, Q)) \land (\neg b \to Q)$$

A critical observation to make with this is that the weakest precondition $\texttt{wp}(W, Q)$ is defined in terms of itself, i.e. it is recursive. To compute this we would need to unroll the loop, i.e. repeatedly substitute $\texttt{wp}(W, Q)$ into itself, infinitely many times. What we need is something which allows us inductively reason about the loop, i.e. we need to introduce a loop invariant. Hence we define a notation

$$\texttt{while } [I] \texttt{ b do s}$$

to remind ourselves, the weakest precondition is defined as a function which is intended to represent a function over a statement s and a postcondition Q such that if our function holds before executing s then Q holds after executing s. Now a trivial approach could be to say

$$\texttt{wp}(\texttt{while } [I] \texttt{ b do s}, Q) = I$$

but we run into the issue that while the inductive invariant remains valid throughout the loop execution, but it is not the case that I implies Q after the loop terminates. Here our two conditions come into play

- Preservation - We need to ensure that whenever the loop guard holds $I \wedge b$, executing the body s must preserve or re-establish the invariant I. This is expressed as

$$(I \wedge b) \rightarrow \texttt{wp}(s, I)$$

- Termination - When the loop exists so $I \wedge \neg b$ (i.e. when the guard b is false) then the invariant I must imply our postcondition Q. This is expressed as

$$\underbrace{(I \wedge \neg b)}_{\text{if our loop exists}} \rightarrow \underbrace{Q}_{\text{the PC holds}}$$

We can package these two conditions together to form our verification conditions for loops.

$$\texttt{vc}(\texttt{while } [I] \texttt{ b do s}, Q) = ((I \wedge b) \rightarrow \texttt{wp}(s, I)) \wedge \texttt{vc}(s, I) \wedge ((I \wedge \neg b) \rightarrow Q) \tag{11}$$

Breaking this down it implies 3 main things

1. Preservation - The loop body s preserves the invariant I whenever the loop guard b holds.
2. Body VCs - The verification conditions for the loop body s with respect to the invariant I must also hold.
3. Termination - When the loop exits (i.e. when b is false), the invariant I must imply the postcondition Q.

One important note here is that the assertion

$$\texttt{wp}(\texttt{while } [I] \texttt{ b do s}, Q) = I$$

is only unsound if we do not also include the verification conditions, if we do include them then it is sound.

A sort of mini summary of the weakest precondition calculus is as follows:

- The weakest precondition $\texttt{wp}(s, Q)$ represents the minimal or least restrictive condition or requirement on the initial state of our program to guarantee that *after execution* the postcondition Q holds.
- Weakest preconditions are derived through backward reasoning, starting from the postcondition Q and working backwards through the program statements to determine the least restrictive condition on the initial state.

3.1.3 Verification Condition

So as a reminder, our goal in assuring our program does what it wants is that we reduce an annotated function i.e. a function with preconditions, postconditions and loop invariants to a set of First Order Logic (FOL) formulas - called *Verification Conditions* (VCs) - such that the validity of these formulas implies the correctness of our program. To be more granular what we want is some way to formally express

some starting assumptions P, *guarantees* after executing statement s, we end up in a state satisfying Q

Now we know that the *starting assumptions* are our preconditions P. Furthermore, if we want a *guarantee* that's equivalent to saying we want our starting conditions to *imply* validity of the post condition Q after executing statement s. Finally we know that the weakest precondition $\texttt{wp}(s, Q)$ is the least restrictive condition on the initial state to guarantee that Q holds after executing s. Hence we can express our verification condition as

$$P \rightarrow \texttt{wp}(s, Q)$$

The intuitive idea here being that since we know that

$$\mathtt{wp}(s, Q) \to P'$$

Where P' is some condition we know guarantees Q after executing s, then if

$$P \to P'$$

We know P is strong enough to guarantee Q after executing s.

Formally we break this down as follows:

- Skip
$$\mathtt{vc}(\mathtt{skip}, Q) = \top$$
- Assignment
$$\mathtt{vc}(x := e, Q) = \top$$
- Sequence
$$\mathtt{vc}(s_1;\ s_2, Q) = \mathtt{vc}(s_1, \mathtt{wp}(s_2, Q)) \wedge \mathtt{vc}(s_2, Q)$$
- If
$$\mathtt{vc}(\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, Q) = \mathtt{vc}(s_1, Q) \wedge \mathtt{vc}(s_2, Q)$$
- While
$$\mathtt{vc}(\mathtt{while}\ [I]\ b\ \mathtt{do}\ s, Q) = \underbrace{((I \wedge b) \to \mathtt{wp}(s, I))}_{\text{Preservation}} \wedge \underbrace{\mathtt{vc}(s, I)}_{\text{Body VC's}} \wedge \underbrace{((I \wedge \neg b) \to Q)}_{\text{Termination}}$$
- Overall VC
$$\mathtt{vc}(\{P\}\ s\ \{Q\}) = (P \to \mathtt{wp}(s, Q)) \wedge \mathtt{vc}(s, Q)$$

As an example Let's look at some lean code, we'll start with the hoare logic for the while loop

```
inductive Stmt : Type where
  | whileDo : (B : State → Prop) → (S : Stmt) → Stmt

theorem while_intro (P) {B S}
  (h : {* fun s ↦ P s ∧ B s *} (S) {* P *})
: {* P *} (Stmt.whileDo B S) {* fun s ↦ P s ∧ ¬ B s *}

theorem while_intro' {B P Q S} (I)
  (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})
  (hP : ∀s, P s → I s)
  (hQ : ∀s, ¬ B s → I s → Q s)
: {* P *} (Stmt.whileDo B S) {* Q *}
```

this should all be pretty familiar for the most part, so `while_intro` is the basic while rule, and `while_intro'` is the invariant based while rule.

```
def Stmt.invWhileDo (I B : State → Prop) (S : Stmt) : Stmt :=
  Stmt.whileDo B S

theorem invWhile_intro {B I Q S}
  (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})  -- (I ∧ B) → wp(S, I)
  (hQ : ∀s, ¬ B s → I s → Q s)                -- (I ∧ ¬B) → Q
: {* I *} (Stmt.invWhileDo I B S) {* Q *} :=
  while_intro' I hS (by aesop) hQ

theorem invWhile_intro' {B I P Q S}
```

```
  (hP : ∀s, P s → I s)                          -- P → I
  (hS : {* fun s ↦ I s ∧ B s *} (S) {* I *})    -- (I ∧ B) → wp(S, I)
  (hQ : ∀s, ¬ B s → I s → Q s)                  -- (I ∧ ¬B) → Q
: {* P *} (Stmt.invWhileDo I B S) {* Q *} :=
  while_intro' I hS hP hQ
```

We define a constant `invWhileDo` to represent a while loop with an invariant, and we define two theorems `invWhile_intro` and `invWhile_intro'` which are just wrappers around `while_intro'` to make it easier to apply.

```
def matchPartialHoare : Expr → Option (Expr × Expr × Expr)
  | Expr.app (Expr.app (Expr.app (Expr.const'' PartialHoare _) P) S) Q =>
    Option.some (P, S, Q)
  | _ =>
    Option.none

partial def vcg : TacticM Unit := do
  let goals ← getUnsolvedGoals
  if goals.length != 0 then
    let target ← getMainTarget
    match matchPartialHoare target with
    | Option.none => return
    | Option.some (P, S, Q) =>
      --- other cases omitted for brevity
      else if Expr.isAppOfArity S'' Stmt.invWhileDo 3 then
        if Expr.isMVar P then
          -- apply PartialHoare.invWhile_intro
        else
          -- apply PartialHoare.invWhile_intro'
      else
        failure

elab "vcg" : tactic => vcg
```

Finally we have our actual verification condition generator `vcg`, which is a tactic which recursively applies the appropriate hoare logic rules based on the structure of the statement s. I wanted to focus here particularly on the while loop case. We can break it down line by line:

1. Checking if the statement s is a while loop with an invariant. To check this we use `Expr.isAppOfArity` to check if the expression is an application of `Stmt.invWhileDo` with 3 arguments (the invariant, the guard and the body).
2. Checking if the precondition P is a metavariable (i.e. not specified).
   - Case 1: P is a metavariable - we apply the `PartialHoare.invWhile_intro` rule which only requires the invariant to be specified. Our subgoals become

$$\underbrace{\{I \wedge b\}\ s\ \{I\}}_{\mathrm{vc}(S,I)\ \wedge\ (I\ \wedge\ B)\ \to\ \mathrm{wp}(S,I)} \quad \text{and} \quad \underbrace{\forall s,\ \neg b \to I\ s \to Q\ s}_{(I\ \wedge\ \neg B)\ \to\ Q}$$

   - Case 2: P is not a metavariable - we apply the `PartialHoare.invWhile_intro'` rule which requires both the precondition and the invariant to be specified. Our subgoals become

$$\underbrace{\{I \wedge b\}\ s\ \{I\}}_{\mathrm{vc}(S,I)\ \wedge\ (I\ \wedge\ B)\ \to\ \mathrm{wp}(S,I)} \quad \text{and} \quad \underbrace{\forall s,\ P\ s \to I\ s}_{P\ \to\ I} \quad \wedge \quad \underbrace{\forall s,\ \neg b \to I\ s \to Q\ s}_{(I\ \wedge\ \neg B)\ \to\ Q}$$

Example 3.1. Simple Assignment
Let's consider the following

```
// @pre x ⩾ 0
x := x + 1;
// @post x ⩾ 1
```

The verification condition we need to prove is

$$\{x \geqslant 0\}\ (x := x + 1)\ \{x \geqslant 1\} : x \geqslant 0 \to \mathsf{wp}(x := x + 1, x \geqslant 1)$$

computing this we get the following

$$\begin{aligned}\mathsf{wp}(x := x + 1, x \geqslant 1) &\Leftrightarrow (\lambda x.\ x \geqslant 1)(x + 1)\\ &\Leftrightarrow x + 1 \geqslant 1\\ &\Leftrightarrow x \geqslant 0\end{aligned}$$

Example 3.2. If Statement
Let's consider the following

```
// @pre P
x = y + 1;
if x > 0 then
    z = 1;
else
    z = -1;
// @post Q
```

Let's consider a few situation

- What is $\mathsf{wp}(s, z > 0)$ where s is the if statement?

  $$\begin{aligned}\mathsf{wp}(s, z > 0) &\Leftrightarrow (x > 0 \to \mathsf{wp}(z := 1, z > 0)) \wedge (\neg(x > 0) \to \mathsf{wp}(z := -1, z > 0))\\ &\Leftrightarrow (x > 0 \to (z > 0)[1/z]) \wedge (\neg(x > 0) \to (z > 0)[-1/z])\\ &\Leftrightarrow (x > 0 \to 1 > 0) \wedge (\neg(x > 0) \to -1 > 0)\\ &\Leftrightarrow (x > 0 \to \top) \wedge (\neg(x > 0) \to \bot)\\ &\Leftrightarrow \top \wedge (x > 0)\\ &\Leftrightarrow x > 0\end{aligned}$$

  From the assignment before the if statement we have

  $$x := y + 1 \to \mathsf{wp}(x := y + 1, x > 0) \equiv (x > 0)[y + 1/x] \equiv y + 1 > 0 \equiv y > -1 \equiv y \geqslant 0$$

  hence

  $$\mathsf{wp}(s, z > 0) \equiv y \geqslant 0$$

- What is $\mathsf{wp}(s, z \leqslant 0)$ where s is the if statement?

  $$\begin{aligned}\mathsf{wp}(s, z \leqslant 0) &\Leftrightarrow (x > 0 \to \mathsf{wp}(z := 1, z \leqslant 0)) \wedge (\neg(x > 0) \to \mathsf{wp}(z := -1, z \leqslant 0))\\ &\Leftrightarrow (x > 0 \to (z \leqslant 0)[1/z]) \wedge (\neg(x > 0) \to (z \leqslant 0)[-1/z])\\ &\Leftrightarrow (x > 0 \to 1 \leqslant 0) \wedge (\neg(x > 0) \to -1 \leqslant 0)\\ &\Leftrightarrow (x > 0 \to \bot) \wedge (\neg(x > 0) \to \top)\\ &\Leftrightarrow \bot \wedge (\neg(x > 0) \to \top)\\ &\Leftrightarrow x \leqslant 0\end{aligned}$$

  From the assignment before the if statement we have

  $$x := y + 1 \to \mathsf{wp}(x := y + 1, x \leqslant 0) \equiv (x \leqslant 0)[y + 1/x] \equiv y + 1 \leqslant 0 \equiv y \leqslant -1$$

  hence

  $$\mathsf{wp}(s, z \leqslant 0) \equiv y \leqslant 1$$

Another important equivalence used here is

$$A \to B \equiv \neg A \vee B$$

which provides us with

$$\neg(x > 0) \to \top \equiv x \leqslant 0 \vee \bot \equiv x \leqslant 0$$

- Can we solve the following hoare triple?

  $$\{-1 \leqslant y\}\ s\ \{z > 0\}$$

  For the case $Q = z > 0$ our weakest precondition is $y > 1$. As $-1 \leqslant y \not\to y > 1$ this is not valid.

- Can we solve the following hoare triple?

  $$\{y \leqslant 1\}\ s\ \{z \leqslant 0\}$$

  For the case $Q = z \leqslant 0$ our weakest precondition is $y \leqslant 1$. As $y \leqslant 1 \to y \leqslant 1$ this is valid.

Example 3.3. While loops
Let's consider

```
// @pre x ⩽ 0
while [x <= 6] (x <= 5) do
    x = x + 1;
// @post x = 6
```

Breaking this down we have

- Invariant - $I \equiv x \leqslant 6$
- Guard - $b \equiv x \leqslant 5$
- Body - $s \equiv x := x + 1$
- Postcondition - $Q \equiv x = 6$

Applying our verification condition for loops we have

$$\begin{aligned} \mathsf{vc}(\texttt{while}\ [I]\ b\ \texttt{do}\ s, Q) = {} & ((x \leqslant 6 \wedge x \leqslant 5) \to \mathsf{wp}(x := x + 1, x \leqslant 6)) && (1)\\ & \wedge \mathsf{vc}(x := x + 1, x \leqslant 6) && (2)\\ & \wedge ((x \leqslant 6 \wedge \neg(x \leqslant 5)) \to x = 6) && (3) \end{aligned}$$

Breaking this down we have

1. Preservation

$$\begin{aligned} &((x \leqslant 6 \wedge x \leqslant 5) \to \mathsf{wp}(x := x + 1, x \leqslant 6))\\ &\qquad \equiv ((I \wedge b) \to (x \leqslant 6)[x + 1/x])\\ &\qquad \equiv ((I \wedge b) \to x + 1 \leqslant 6)\\ &\qquad \equiv ((x \leqslant 6 \wedge x \leqslant 5) \to x \leqslant 5)\\ &\qquad \equiv \top \end{aligned}$$

2. Body VCs

$$\mathsf{vc}(x := x + 1, x \leqslant 6) \equiv \top \qquad \text{(assignment has no VCs)}$$

3. Termination

$$\begin{aligned} &((x \leqslant 6 \wedge \neg(x \leqslant 5)) \to x = 6)\\ &\qquad \equiv (x \leqslant 6 \wedge x > 5) \to x = 6\\ &\qquad \equiv (x = 6) \to x = 6 && \text{(as } x \text{ is an integer)}\\ &\qquad \equiv \top \end{aligned}$$

Thus we have

$$\mathsf{vc}(\texttt{while}\ [\mathrm{I}]\ \mathrm{b}\ \texttt{do}\ \mathrm{s}, \mathrm{Q}) \equiv \top \wedge \top \wedge \top \equiv \top$$

importantly we can observe that

$$(\underbrace{x \leqslant 6}_{\text{Invariant I}} \wedge \underbrace{\neg(x \leqslant 5)}_{\text{loop exit } \neg b}) \quad \underbrace{\rightarrow x = 6}_{\text{implies postcondition Q}}$$

Hence

$$\mathsf{vc}(\{x \leqslant 0\}\ \texttt{while}\ [x \leqslant 6]\ (x \leqslant 5)\ \texttt{do}\ (x := x + 1)\ \{x = 6\})$$

is valid if

$$(x \leqslant 0) \rightarrow (x \leqslant 6) \wedge \mathsf{vc}(\texttt{while}\ [x \leqslant 6]\ (x \leqslant 5)\ \texttt{do}\ (x := x + 1), x = 6)$$

which we can see is the case as $x \leqslant 0 \rightarrow x \leqslant 6$ is valid. Hence our overall verification condition is valid.

Example 3.4. Nested While loops
Let's consider the following code

```
1  i   = 1;
2  sum = 0;
3  while i <= n do [sum >= 0] {
4      j = 1;
5      while j <= i do [sum >= 0 and j >= 0] {
6          sum = sum + j;
7          j   = j + 1;
8          i   = i + 1;
9      }
10 }
```

Let's start off by defining some notation for the inner loop $\mathrm{L_{in}}$ and the outer loop $\mathrm{L_{out}}$.

$$\mathrm{L_{in}} \triangleq \begin{cases} \mathrm{I_{in}} := [\mathsf{sum} \geqslant 0 \wedge j \geqslant 0], \\ \mathrm{b_{in}} := (j \leqslant i), \\ \mathrm{s_{in}} := (\mathsf{sum} := \mathsf{sum} + j;\ j := j + 1) \end{cases}$$

$$\mathrm{L_{out}} \triangleq \begin{cases} \mathrm{I_{out}} := [\mathsf{sum} \geqslant 0], \\ \mathrm{b_{out}} := (i \leqslant n), \\ \mathrm{s_{out}} := (j := 1;\ \mathrm{L_{in}};\ i := i + 1) \end{cases}$$

Now Let's say we are given the target postcondition of

$$\mathrm{Q} \equiv \mathsf{sum} \geqslant 0$$

Starting with the outer loop, we can compute the verification conditions as follows

$$\begin{aligned} \mathsf{vc}(\mathrm{L_{out}}, \mathrm{Q}) = ((\mathrm{I_{out}} \wedge \mathrm{b_{out}}) \rightarrow \mathsf{wp}(\mathrm{s_{out}}, \mathrm{I_{out}})) \quad (1) \\ \wedge\, \mathsf{vc}(\mathrm{s_{out}}, \mathrm{I_{out}}) \quad (2) \\ \wedge\, ((\mathrm{I_{out}} \wedge \neg \mathrm{b_{out}}) \rightarrow \mathrm{Q}) \quad (3) \end{aligned}$$

Breaking this down we have

1. Preservation

$$\begin{aligned} &((\mathsf{sum} \geqslant 0 \wedge i \leqslant n) \rightarrow \mathsf{wp}(j := 1;\ \mathrm{L_{in}};\ i := i + 1, \mathsf{sum} \geqslant 0)) \\ &\quad \equiv ((\mathrm{I_{out}} \wedge \mathrm{b_{out}}) \rightarrow \mathsf{wp}(j := 1, \mathsf{wp}(\mathrm{L_{in}}, \mathsf{wp}(i := i + 1, \mathsf{sum} \geqslant 0)))) \\ &\quad \equiv ((\mathrm{I_{out}} \wedge \mathrm{b_{out}}) \rightarrow \mathsf{wp}(j := 1, \mathsf{wp}(\mathrm{L_{in}}, \mathsf{sum} \geqslant 0))) \\ &\quad \equiv ((\mathrm{I_{out}} \wedge \mathrm{b_{out}}) \rightarrow \mathsf{wp}(j := 1, \mathrm{I_{in}})) \\ &\quad \equiv ((\mathrm{I_{out}} \wedge \mathrm{b_{out}}) \rightarrow (\mathsf{sum} \geqslant 0 \wedge 1 \geqslant 0)) \\ &\quad \equiv ((\mathsf{sum} \geqslant 0 \wedge i \leqslant n) \rightarrow (\mathsf{sum} \geqslant 0)) \end{aligned}$$

2. Body VCs

$$\begin{aligned}\mathsf{vc}(s_{\text{out}}, I_{\text{out}}) &\equiv \mathsf{vc}(j := 1;\ L_{\text{in}};\ i := i + 1, \mathsf{sum} \geqslant 0)\\ &\equiv \mathsf{vc}(j := 1, \mathsf{vc}(L_{\text{in}}, \mathsf{vc}(i := i + 1, \mathsf{sum} \geqslant 0)))\\ &\equiv \mathsf{vc}(j := 1, \mathsf{vc}(L_{\text{in}}, \mathsf{sum} \geqslant 0))\\ &\equiv \mathsf{vc}(L_{\text{in}}, \mathsf{sum} \geqslant 0) \qquad \text{(assignment has no VCs)}\end{aligned}$$

3. Termination

$$\begin{aligned}((\mathsf{sum} \geqslant 0 \land \neg(i \leqslant n)) &\to \mathsf{sum} \geqslant 0)\\ &\equiv ((\mathsf{sum} \geqslant 0 \land i > n) \to \mathsf{sum} \geqslant 0)\\ &\equiv \top\end{aligned}$$

Similarly for the inner loop

$$\begin{aligned}\mathsf{vc}(L_{\text{in}}, \mathsf{sum} \geqslant 0) = &((I_{\text{in}} \land b_{\text{in}}) \to \mathsf{wp}(s_{\text{in}}, I_{\text{in}})) &(1)\\ &\land \mathsf{vc}(s_{\text{in}}, I_{\text{in}}) &(2)\\ &\land ((I_{\text{in}} \land \neg b_{\text{in}}) \to \mathsf{sum} \geqslant 0) &(3)\end{aligned}$$

Breaking this down we have

1. Preservation

$$\begin{aligned}((\mathsf{sum} \geqslant 0 \land j \geqslant 0 \land j \leqslant i) &\to \mathsf{wp}(\mathsf{sum} := \mathsf{sum} + j;\ j := j + 1, \mathsf{sum} \geqslant 0))\\ &\equiv ((I_{\text{in}} \land b_{\text{in}}) \to \mathsf{wp}(\mathsf{sum} := \mathsf{sum} + j, \mathsf{wp}(j := j + 1, \mathsf{sum} \geqslant 0)))\\ &\equiv ((I_{\text{in}} \land b_{\text{in}}) \to \mathsf{wp}(\mathsf{sum} := \mathsf{sum} + j, \mathsf{sum} \geqslant 0))\\ &\equiv ((I_{\text{in}} \land b_{\text{in}}) \to (\mathsf{sum} \geqslant 0)[\mathsf{sum} + j/\mathsf{sum}])\\ &\equiv ((\mathsf{sum} \geqslant 0 \land j \geqslant 0 \land j \leqslant i) \to (\mathsf{sum} + j \geqslant 0))\\ &\equiv \top\end{aligned}$$

2. Body VCs

$$\begin{aligned}\mathsf{vc}(s_{\text{in}}, I_{\text{in}}) &\equiv \mathsf{vc}(\mathsf{sum} := \mathsf{sum} + j;\ j := j + 1, \mathsf{sum} \geqslant 0 \land j \geqslant 0)\\ &\equiv \mathsf{vc}(\mathsf{sum} := \mathsf{sum} + j, \mathsf{vc}(j := j + 1, \mathsf{sum} \geqslant 0 \land j \geqslant 0))\\ &\equiv \mathsf{vc}(\mathsf{sum} := \mathsf{sum} + j, \mathsf{sum} \geqslant 0 \land j + 1 \geqslant 0)\\ &\equiv \top \qquad \text{(assignment has no VCs)}\end{aligned}$$

3. Termination

$$\begin{aligned}((\mathsf{sum} \geqslant 0 \land j \geqslant 0 \land \neg(j \leqslant i)) &\to \mathsf{sum} \geqslant 0)\\ &\equiv ((\mathsf{sum} \geqslant 0 \land j \geqslant 0 \land j > i) \to \mathsf{sum} \geqslant 0)\\ &\equiv \top\end{aligned}$$

Thus we have

$$\mathsf{vc}(L_{\text{in}}, \mathsf{sum} \geqslant 0) \equiv \top \land \top \land \top \equiv \top$$

and hence

$$\mathsf{vc}(L_{\text{out}}, Q) \equiv \top \land \top \land \top \equiv \top$$

3.2 Assertions

We can introduce 3 new statements to our language

- `assert F` - This statement checks that the condition F holds at that point in the program. If F is true, the program continues executing normally. If F is false, the program raises an assertion failure, which typically halts execution and may provide diagnostic information about the failure.

- `assume F` - This statement is used to inform the verification tool that we are assuming the condition F holds at that point in the program. It does not perform any runtime check; instead, it serves as a hint to the verifier to consider F as a given fact for subsequent analysis. If F is false, the behavior of the program is undefined, as the assumption is violated.
- `x := havoc` - This statement is used to model non-deterministic behavior in a program. When the program reaches a havoc statement, the variable x is assigned an arbitrary value, effectively "resetting" it to an unknown state. This is useful in verification to represent situations where the value of x can be anything, allowing the analysis to consider all possible values for x in subsequent computations.

3.2.1 Semantics

We can define the semantics of these statements as follows

$$\frac{\sigma \models F}{\langle \texttt{assert } F, \sigma\rangle \to^* \sigma}\ \textsc{Assert-}\top \qquad \frac{\sigma \not\models F}{\langle \texttt{assert } F, \sigma\rangle \to^* \texttt{fail}}\ \textsc{Assert-}\bot \qquad \frac{\sigma \models F}{\langle \texttt{assume } F, \sigma\rangle \to^* \sigma}\ \textsc{Assume-}\top$$

$$\frac{\sigma \not\models F}{\langle \texttt{assume } F, \sigma\rangle \to^* \texttt{stuck}}\ \textsc{Assume-}\bot \qquad \frac{v \in \mathbb{Z}}{\langle x := \texttt{havoc()}, \sigma\rangle \to^* \sigma[x \mapsto v]}\ \textsc{Havoc}$$

We introduce a new state `fail` to represent assertion failures, and a new state `stuck` to represent the case where an assumption is violated. Notably we can observe two things

- Assert behaves similarly to Skip in that if the assertion holds we continue executing normally, otherwise we raise an error.
- Assume Similarly behaves like Skip if the assumption holds, but if it doesn't hold we get stuck.

3.2.2 Hoare Logic Rules

We can define the following hoare logic rules for these statements

$$\frac{P \to F}{\{P\}\ \texttt{assert } F\ \{P \wedge F\}}\ \textsc{Assert} \qquad \frac{}{\{F\}\ \texttt{assume } F\ \{P \wedge F\}}\ \textsc{Assume} \qquad \frac{}{\{\forall y.\ Q[y \mapsto x]\}\ x := \texttt{havoc()}\ \{Q\}}\ \textsc{Havoc}$$

3.2.3 Verification Conditions

We can define the following additional weakest precondition rules for these statements

- Assert
$$\texttt{wp}(\texttt{assert } F, Q) = F \wedge Q \quad \texttt{vc}(\texttt{assert } F, Q) = \top$$
- Assume
$$\texttt{wp}(\texttt{assume } F, Q) = F \to Q \quad \texttt{vc}(\texttt{assume } F, Q) = \top$$

Example 3.5. Assertions and Assumptions in practice
Let's consider a statement s and s' and some transformation $s \to s'$ such that the following is valid

$$\{P\}\ s\ \{Q\} \iff \{\top\}\ s'\{\top\}$$

Let's take s' to be

$$s' \equiv \texttt{assume } P;\ s;\ \texttt{assert } Q$$

Now computing the weakest precondition over the sequences we have

$$\begin{aligned}
\texttt{wp}(s', \top) &\equiv \texttt{wp}(\texttt{assume } P;\ s;\ \texttt{assert } Q, \top) \\
&\equiv \texttt{wp}(\texttt{assume } P, \texttt{wp}(s;\ \texttt{assert } Q, \top)) \\
&\equiv P \to \texttt{wp}(s;\ \texttt{assert } Q, \top) \\
&\equiv P \to \texttt{wp}(s, \texttt{wp}(\texttt{assert } Q, \top)) \\
&\equiv P \to \texttt{wp}(s, Q \wedge \top) \\
&\equiv P \to \texttt{wp}(s, Q)
\end{aligned}$$

We also want to copute the verification condition for s'

$$\begin{aligned}
\mathtt{vc}(s', \top) &\equiv \mathtt{vc}(\mathtt{assume}\ P;\ s;\ \mathtt{assert}\ Q, \top) \\
&\equiv \mathtt{vc}(\mathtt{assert}\ Q, \top) \wedge \mathtt{vc}(\mathtt{assume}\ P, \mathtt{wp}(\mathtt{assert}\ Q), \top) \\
&\equiv \underbrace{\mathtt{vc}(\mathtt{assert}\ Q, \top)}_{\top} \wedge \mathtt{vc}(s, \underbrace{\mathtt{wp}(\mathtt{assert}\ Q, \top)}_{Q}) \wedge \underbrace{\mathtt{vc}(\mathtt{assume}\ P, \mathtt{wp}(s, \mathtt{wp}(\mathtt{assert}\ Q, \top)))}_{\top} \\
&\equiv \mathtt{vc}(s, Q)
\end{aligned}$$

Hence we have

$$\begin{aligned}
\mathtt{vc}(\{\top\}\, s'\, \{\top\}) &\equiv \mathtt{vc}(s', \top) \wedge \top \to \mathtt{wp}(s', \top) \\
&\equiv \mathtt{vc}(s, Q) \wedge \top \to (\top \to (P \to \mathtt{wp}(s, Q))) \\
&\equiv \mathtt{vc}(s, Q) \wedge P \to \mathtt{wp}(s, Q) \\
&\equiv \mathtt{vc}(\{P\}\, s\, \{Q\})
\end{aligned}$$

3.3 Functions

3.3.1 Grammar

We can extend our grammar to include functions as follows

$$\begin{aligned}
\text{Prog} = p &::= f+ && (\geqslant 1 \text{ function in the Program}) \\
\text{Fun} = f &::= \mathtt{fun}\ f(x_1, \ldots, x_n)\, \{s;\ \mathtt{return}\ e\} \\
\text{Stmt} = s &::= f(e_1, \ldots, e_n) \mid \ldots && (\text{function call statement})
\end{aligned}$$

3.3.2 Handling functions

To handle functions similar to the case of loops having to provide an invariant, we will need to provide a function specification for each function in the program. A function specification consists of a precondition and a postcondition. Formally we say that

- Precondition - A predicate P over the function's parameters and possibly some global state that must hold true before the function is called.
- Postcondition - A predicate Q over the function's return value, parameters and possibly some global state that must hold true after the function has executed, assuming the precondition was satisfied.

The main question at hand becomes now, how do we verify a triple in the following form

$$\{P\}\, y := f(\overline{a})\, \{Q\} \quad \overline{a} = a_1, \ldots, a_n$$

Abstractly the verification conditions for the function are defined as follows

- `assert` $\mathtt{Pre}_f(\overline{a})$ - We need to ensure that the precondition of the function holds before the function is called under the substitution of the arguments we are calling with. This is done by adding an assertion for the precondition at the point of the function call.
- `assume` $\mathtt{Post}_f(\overline{a}, \mathtt{ret}_f)$ - We need to ensure that the postcondition of the function holds after the function has executed. This is done by adding an assumption for the postcondition at the point immediately after the function call, where `ret` is a special variable representing the return value of the function.

With this in mind we can redefine our function callsite as follows

```
assert Pre_f(a_1, ..., a_n);
assume Post_f(a_1, ..., a_n, ret_f);
y = ret_f;
```

Two important notes about $\mathtt{ret}_f$

- Freshness - The variable $\texttt{ret}_f$ must be fresh, i.e. it must not clash with any other variable in the program. This is as we want to represent a new return value independent of any other variable in the program.
- Assignment - We assign the variable to y as `assume` does not change the state of the program, it simply represents an assumption.

The triple for the body of a function is then represented as follows

$$\{\texttt{Pre}_f(\overline{x})\}\ s;\ \texttt{ret}_f := e\ \{\texttt{Post}_f(\overline{x}, e)\}$$

or equivalently as code

```
method f(x_1: int, ..., x_n: int) returns (ret_f: int)
  requires Pre_f(x_1, ..., x_n)
  ensures Post_f(x_1, ..., x_n, ret_f)
{
    s;
    return e;
}
```

3.3.3 Modularity

An important concept to see here is the different way we consider the pre and post condition depending on if we are verifying the function definition or the function callsite.

- Definition - When verifying the function definition we *assume* the precondition holds at the start of the function, and we *assert* that the postcondition holds at the end of the function.
- Callsite - When verifying the function callsite we instead instead *assert* that the precondition holds, or more concretely we must show that the arguments we are calling the functionw with satisfy the precondition. We then *assume* that the postcondition holds given the assurances we have provided by the precondition.

The key benefit we derive from this manner of verification is that we verify the function bodies once, and then we can use the specifications to verify the callsites without needing to re-verify the function body each time, i.e. we an avoid inlining the function body at each callsite. This is what is meant by **modular verification**.

Another way of understanding this in a more intuitive manner I think is that

- The programmer demonstrates through the assertion of the precondition that the calling arguments are indeed valid for the function to execute correctly, they are in turn then given the assurance that they can assume the postcondition holds after the function call.
- The function implementer in turn has the provided assurance that the precondition will hold when the function is called, as this is the responsibility of the caller. They then must do their part in ensuring that the postcondition i.e. the contract of the function is fulfilled.

3.3.4 Weakest Precondition

We can define the weakest precondition for function calls as follows

$$\texttt{wp}(y := f(\overline{a}), Q) = \texttt{Pre}_f(\overline{a}) \wedge \forall e.\ (\texttt{Post}_f(\overline{a}, e) \rightarrow Q[\texttt{ret}_f \mapsto e])$$

3.3.5 Mutable state and purity

One thing we sort of glossed over here is the idea of how to actually accont for mutating a piece of state. In the above we have implicitly assumed that functions are *pure*, i.e. they do not modify any state outside of their local scope, though naturally we can't guarantee this is always the case. Hence we typically want to specify which variables a function is allowed to modify.
We can introduce a new clause to our function specification called `modifies` which is a set of variables that the function is allowed to modify. Extending our dafny example from before we have

```
method f(x_1: int, ..., x_n: int) returns (ret_f: int)
  modifies (v_1, ..., v_k)
  requires Pre_f(x_1, ..., x_n)
  ensures Post_f(x_1, ..., x_n, ret_f)
{
    s;
    return e;
}
```

The next question naturally becomes, what about the callsite? Here we have a slight reordering

```
assert Pre_f(a_1, ..., a_n);
havoc (v_1, ..., v_k); // ∀i. v_i is fresh
y = ret_f;
assume Post_f(a_1, ..., a_n, ret_f);
```

If we reason about this step by step it translates to

1. ASSERT - We assert that the precondition holds for the arguments we are calling the function with.
2. HAVOC - We havoc all the variables that the function is allowed to modify, this is to represent the fact that we have no idea what values these variables will have after the function call.
3. ASSIGN - We assign the return value to the variable y.
4. ASSUME - We assume that the postcondition holds given the arguments and the return value.

The intuition behind why need to havoc the modified variables before the assignment and assumption is that we need some way to represent the uncertainty of what the variables will be after the function call. In other words havoc is just a way of saying "these variables could be anything", i.e. it models the non-determinstic nature of the function modifying these variables.

The issue being that if we wouldn't havoc them first, we naturally introduce the possibility of unsound reasoning where we end up having stale values for these variables that don't reflect the changes made by the function.

A nicely formal way we talka bout verification when considering side effects is through the frame rule.

$$\frac{\{P\}\ s\ \{Q\} \qquad \text{Mod}(s) \cap \text{Free}(R) = \emptyset}{\{P \wedge R\}\ s\ \{Q \wedge R\}}\ \textsc{Frame}$$

The rule states that if we have our normal triple $\{P\}\ s\ \{Q\}$ and we have our *frame* R representing facts about the state that are not modified by the statement s, then we can extend our triple to include R in both the pre and post conditions. The side condition $\text{Mod}(s) \cap \text{Free}(R) = \emptyset$ ensures that none of the variables that s modifies are free in R, i.e. s does not modify any of the state that R is talking about.

The side condition is precisely why we need to havoc the modified variables at the callsite, as without doing so we could end up in a situation where we have some fact R about a variable that is modified by the function call, leading to unsound reasoning.

3.4 Pointers

3.4.1 Grammar

$$\text{Stmt} = s ::= y := *x\ (\texttt{load}) \mid *x := e\ (\texttt{store}) \mid \dots$$

3.4.2 The heap

We denote the heap μ as a mapping from addresses to values

$$\mu : \text{Address} \to \text{Value}$$

An impication of this being that we employ the theory of arrays to reason about the heap, where the addresses are the indices and the values are the elements at those indices.

3.4.3 Hoare Logic Rules

$$\frac{}{\{P[*x \mapsto \mu[y]]\}\ y := *x\ \{Q\}}\ \textsc{Load} \qquad \frac{}{\{P[\mu \mapsto \mu\langle x \lhd e\rangle]\}\ *x := e\ \{Q\}}\ \textsc{Store}$$

- Load - When loading from a pointer x into a variable y, we need to update the precondition P to reflect that y now holds the value at the address pointed to by x. This is done by substituting $*x$ with $\mu[y]$ in the precondition.
- Store - When storing a value e into the address pointed to by x, we need to update the precondition P to reflect the new state of the heap. This is done by updating the heap mapping μ to reflect that the address x now holds the value e. The notation $\mu\langle x \lhd e\rangle$ represents the updated heap where the value at address x is now e.

4 Automating Hoare Logic: Part II

4.1 Motivation

We can start by considering the following simple program

$$\{x > 0\}\ \underbrace{y := x;}_{s_1}\underbrace{z := x + y}_{s_2}\ \{z > 0\}$$

Should be obvious we use the sequencing rule here

$$\frac{\{P\}\ s_1\ \{R\} \qquad \{R\}\ s_2\ \{Q\}}{\{P\}\ s_1;\ s_2\ \{Q\}}\ \textsc{Seq}$$

We can understand the sequencing rule as requiring us to find a formula $q(x, y, z)$ such that

$$\{x > 0\}\ y := x\ \{q(x, y, z)\} \quad \text{and} \quad \{q(x, y, z)\}\ z := x + y\ \{z > 0\}$$

holds. One thing you might have noticed in the pattern of the sequencing rule is that it that it resembles the structure of a transitive relation

$$\forall a, b, c.\ R(a, b) \land R(b, c) \to R(a, c)$$

Formally we would say that the hoare triple encodes the following constraints on our formula q

1. (From first triple)
$$(x > 0 \land y = x) \to q(x, y, z)$$
2. (From second triple)
$$(q(x, y, z) \land z' = x + y) \to (z' > 0)$$

Example 4.1. Testing a solution
Let's try the candidate solution $q(x, y, z) \equiv x + y > 0$ and see if it satisfies the constraints

1. First Constraint

$$\begin{aligned}(x > 0 \land y = x) &\to (x + y > 0)\\ &\equiv (x > 0 \land y = x) \to (x + x > 0)\\ &\equiv (x > 0 \land y = x) \to (2x > 0)\\ &\equiv \top\end{aligned}$$

2. Second Constraint

$$\begin{aligned}(x + y > 0 \land z' = x + y) &\to (z' > 0)\\ &\equiv (x + y > 0 \land z' = x + y) \to (x + y > 0)\\ &\equiv \top\end{aligned}$$

Example 4.2. While loops
We can start by defining a while loop

$$W \triangleq \texttt{while}\, x < n\, \texttt{do}\, x := x + 1$$

Our Hoare triple we wish to prove is

$$\{x \leqslant n\}\, W\, \{x = n\}$$

The rule we will use is naturally

$$\frac{\{I \land b\}\, s\, \{I\} \qquad P \to I \qquad I \land \neg b \to Q}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ s\ \{Q\}}\ \textsc{While}$$

This rule requires us to find a formula $q(x, n)$ such that

$$\begin{aligned}(x \leqslant n) &\to q(x, n) && \text{(Initialization } P \to I)\\ \{q(x, n) \land x < n\}\, x := x + 1\, &\{q(x, n)\} && \text{(Preservation } \{I \land b\}\ s\ \{I\})\\ (q(x, n) \land \neg(x < n)) &\to (x = n) && \text{(Termination } I \land \neg b \to Q)\end{aligned}$$

Once again we can think of our formula q as representing a relation in first order logic. The constraints the hoare triple then encodes on this relation are as follows

1. (From initialization)
$$(x \leqslant n) \to q(x, n)$$
2. (From preservation)
$$(q(x, n) \land x < n \land x' = x + 1) \to q(x', n)$$
3. (From termination)
$$(q(x, n) \land x \geqslant n) \to (x = n)$$

Again Let's take some candidate solution for $q(x, n)$ and see if it satisfies the constraints. Let's try $q(x, n) \equiv x \leqslant n$

1. First Constraint

$$\begin{aligned}(x \leqslant n) &\to (x \leqslant n)\\ &\equiv \top\end{aligned}$$

2. Second Constraint

$$\begin{aligned}(x \leqslant n \land x < n \land x' = x + 1) &\to (x' \leqslant n)\\ &\equiv (x \leqslant n \land x < n) \to (x + 1 \leqslant n)\\ &\equiv (x \leqslant n \land x < n) \to (x < n)\\ &\equiv \top\end{aligned}$$

3. Third Constraint

$$\begin{aligned}(x \leqslant n \land x \geqslant n) &\to (x = n)\\ &\equiv (x = n) \to (x = n)\\ &\equiv \top\end{aligned}$$

Thus we have found a solution for q that satisfies all the constraints induced by the hoare triple.

4.2 Syntax

A constrained Horn clause is a formula of the form

$$p(x_1, x_2) \land q(x_1, x_2, x_3) \land r(x_1) \land \ldots \land \varphi \to H$$

where

1. Queries - $p, q, r, \ldots$ are predicate symbols representing relations over the variables $x_1, x_2, x_3, \ldots$ called *queries*.
2. φ - A formula in some decidable theory T that doesn't contain any queries, called the *constraint*.
3. H - The head of the clause, which is either a query or $\bot$ (false).
4. Body - The conjunction of the queries and the constraint on the left side of the implication is called the *body* of the clause.
5. Variables - The variables $x_1, x_2, x_3, \ldots$ are *implicitly* universally quantified over the entire clause. That is when we denote something to be a horn clause the following equivalence holds

$$\mathsf{Horn}(\overline{x}, \varphi, H) \triangleq \forall \overline{x}.\ (\text{body} \to H)$$

In the most direct sense a horn clause is described as being a disjunction of literals with at most one positive literal. The implication form we use here is just a more convenient way of representing the same thing. So a classic definite clause

$$(p \land q \land \ldots \land \phi) \to H$$

reads intuitively as: "if all of $p, q, \ldots, \phi$ hold, then H must also hold". The main value of these clauses is that their structure provides useful properties particularly in the context of logic programming and automated reasoning.

Example 4.3. Well formed Horn clause
We can look at some instances of well and ill formed horn clauses

$$(x < n) \to q(x, n) \tag{1}$$
$$q(x, n) \land (x < n) \to \bot \tag{2}$$
$$q(x, y) \land r(x) \to p(x) \tag{3}$$
$$q(x, y) \to p(x + 1, y) \tag{4}$$
$$q(x) \land \neg r(x) \to p(x) \tag{5}$$
$$q(x) \land r(x) \to p(x) \tag{6}$$
$$q(x) \to (x < n) \tag{7}$$

1. Well formed - Has a single positive literal $q(x, n)$ in the head.
2. Well formed - Has no positive literals in the head, i.e. the head is $\bot$. This is also called a *goal clause*.
3. Well formed - Has a single positive literal $p(x)$ in the head.
4. Ill formed - The head $p(x + 1, y)$ is not a valid query as queries can only be predicate symbols applied to variables, not expressions.

   To make it valid we could introduce a fresh variable x' and rewrite the body as follows

   $$q(x, y) \land (x' = x + 1) \to p(x', y)$$

5. Ill formed - The body contains a negated query $\neg r(x)$, which is not allowed in horn clauses.
6. Ill formed - In the implication form the body must be a conjunction of queries and constraints only.

   To make it well formed we could rewrite it as

   $$q(x) \to p(x) \quad r(x) \to p(x)$$

7. Ill formed - The head must be a query or $\bot$, not a constraint.

To make it well formed we could rewrite it as

$$q(x) \land \neg(x < n) \to \bot$$

4.3 Constrained Horn Clauses (CHCs)

We can define a set $\mathbb{C}$ of constrained horn clauses as follows

$$\mathbb{C} \triangleq \{C_1, \ldots, C_n\}$$

Some important properties we can enumerate as follows

- Dependency - We say that a query p *depends* on a query q if there exists $C \in \mathbb{C}$ such that q appears in the body of C and p is the head of C.
- Recursion - We say that $\mathbb{C}$ is *recursive* if at least one predicate directly or indirectly refers to itself, i.e. there exists a sequence of clauses $C_1, \ldots, C_k$ such that the head of C_1 appears in the body of C_2, the head of C_2 appears in the body of C_3, and so on, with the head of C_k appearing in the body of C_1. In other words the induced dependency graph has at least one cycle.

  We can give an example such as
  $$q(x) \land x' = x + 1 \to q(x')$$
  Here the query q is present in both the body and the head of the clause, indicating that q depends on itself, thus making the set of clauses recursive.

  - Non-recursive solving - If the dependency graph of clauses is acyclic we can topologically order the predicates and solve them in a bottom-up manner akin to how we computed weakest preconditions.
  - Recursive solving - If a set of clauses is recursive then in general solving them is undecidable which is to say that it amounts to finding loop invariants. We also say that its equivalent to finding a fixed point i.e. an inductive condition which makes all clauses true simultaneously.

4.4 Semantics

We define a solution Σ as a function which maps queries to formula in the theory T over the same variables. Denoted as a rule we have

$$\frac{\Sigma(p) \land \Sigma(q) \land \ldots \land \varphi \to \Sigma(r)}{\Sigma \models p(x_1, x_2) \land q(x_1, x_2, x_3) \land \ldots \land \varphi \to r(x_1)} \text{ Solution}$$

Here $\Sigma \models C$ expresses that the solution Σ satisfies the horn clause C. More generally we can say that the set of clauses $\mathbb{C}$ is satisfiable if there exists a solution Σ such that $\Sigma \models C$ for all $C \in \mathbb{C}$. Formally

$$\mathbb{C} \text{ is satisfiable } \iff \exists \Sigma.\, \forall C \in \mathbb{C}.\, \Sigma \models C$$

Example 4.4. Examining solutions
Let's consider the following set of horn clauses

$$x = 1 \to p(x) \tag{1}$$
$$p(x) \land x' = x + 1 \to p(x') \tag{2}$$
$$p(x) \to 0 \leqslant x$$

Now let's consider the following solution

$$\Sigma(p) \equiv x = 1$$

Substituing this into clause 2 we have

$$(x = 1 \land x' = x + 1) \to (x' = 1)$$

clearly this is not valid as for $x = 1$ we have $x' = 2$ which does not satisfy the head of the clause. Hence this solution does not satisfy all the clauses in the set. Looking at another solution

$$\Sigma(p) \equiv x \geqslant 1$$

Substituing into the clauses we have

1. Clause 1
$$(x = 1) \to (x \geqslant 1)$$
which is valid.

2. Clause 2
$$(x \geqslant 1 \land x' = x + 1) \to (x' \geqslant 1)$$
which is also valid.

3. Clause 3
$$(x \geqslant 1) \to (0 \leqslant x)$$
which is valid as well.

Thus we have found a solution Σ that satisfies all the clauses in the set.

4.5 From WP to Horn

- Assignment - For an assignment we substitute e for x in the query and produce no extra clauses.
$$\mathtt{wp}(x := e, p(\overline{x})) \triangleq p(\overline{x})[x \mapsto e]$$

- Sequential composition - Here this mirrors our rule as before
$$\mathtt{wp}(s_1;\ s_2, p(\overline{x})) \triangleq \mathtt{wp}(s_1, \mathtt{wp}(s_2, p(\overline{x})))$$

- Conditional - We introduce a new query q with the appropriate constraints, thus
$$\mathtt{wp}(\mathtt{if}\, b\ \mathtt{then}\, s_1\ \mathtt{else}\, s_2, p(\overline{x})) \triangleq q(\overline{x})$$
along with the clauses
$$q(\overline{x}) \land b \to \mathtt{wp}(s_1, p(\overline{x}))$$
$$q(\overline{x}) \land \neg b \to \mathtt{wp}(s_2, p(\overline{x}))$$

- While loop - Similar to the conditional we introduce a new query q representing the invariant, thus
$$\mathtt{wp}(\mathtt{while}\, b\ \mathtt{do}\, s, p(\overline{x})) \triangleq q(\overline{x})$$
along with the clauses
$$p(\overline{x}) \to q(\overline{x}) \qquad \text{(Initialization)}$$
$$q(\overline{x}) \land b \to \mathtt{wp}(s, q(\overline{x})) \qquad \text{(Preservation)}$$
$$q(\overline{x}) \land \neg b \to p(\overline{x}) \qquad \text{(Termination)}$$

Example 4.5. Assignment
Consider the following
$$\mathtt{wp}(x := x + 1, p(x, y))$$
applying the assignment rule we have
$$p(x, y)[x \mapsto x + 1] \equiv p(x + 1, y)$$

Example 4.6. Conditional & Sequence
Consider the following statement s
$$x := y + 1;\ \mathtt{if}\, x > 0\ \mathtt{then}\, z := 1\ \mathtt{else}\, z := -1$$
Now Let's try and solve
$$\mathtt{wp}(s, p(x, y, z))$$

Expanding out the sequential composition we have

$$\mathtt{wp}(x := y + 1, \mathtt{wp}(\mathtt{if}\, x > 0\ \mathtt{then}\, z := 1\ \mathtt{else}\, z := -1, p(x, y, z)))$$

Solving the inner conditional first we introduce a new query $q(x, y, z)$ and produce the clauses

$$q(x, y, z) \land (x > 0) \to p(x, y, z)[z \mapsto 1]$$
$$q(x, y, z) \land \neg(x > 0) \to p(x, y, z)[z \mapsto -1]$$

Now solving the outer assignment we have

$$\mathtt{wp}(x := y + 1, q(x, y, z)) \equiv q(x, y, z)[x \mapsto y + 1]$$

Thus our final result is

$$\mathtt{wp}(s, p(x, y, z)) \equiv q(y + 1, y, z)$$

Example 4.7. While loop
Let's use the example from the motivation

$$\{x \leqslant n\}\ \mathtt{while}\, x < n\ \mathtt{do}\, x := x + 1\ \{x = n\}$$

Our goal is to solve

$$\mathtt{wp}(\mathtt{while}\, x < n\ \mathtt{do}\, x := x + 1, p(x, n))$$

We again introduce a new query $q(x, n)$ and produce the clauses

$$(x \leqslant n) \to q(x, n) \qquad \text{(Initialization)}$$
$$q(x, n) \land (x < n) \to q(x + 1, n) \qquad \text{(Preservation)}$$
$$q(x, n) \land x \geqslant n \to p(x, n) \qquad \text{(Termination)}$$

Where

$$p(x, n) \to x = n$$

What we can note here is that - after some rearrangement - these are precisely the same constraints we derived in the motivation example for the while loop.

4.6 Normalization

What we noticed in the previous example is that despite applying the horn clause translation mechanically we ended up with clauses that were not in the standard form we defined earlier. To fix this we can apply a normalization step to ensure all clauses are in the correct form. The central normalization rule we apply is as follows. Given a clause of the form

$$p(e_1, e_2) \land \ldots \land \varphi \to r(e_3)$$

we can introduce fresh variables x_1, x_2, x_3 and rewrite the clause as

$$p(x_1, x_2) \land \ldots \land \varphi \land (x_1 = e_1) \land (x_2 = e_2) \land (x_3 = e_3) \to r(x_3)$$

4.7 Verifying Hoare Triples

Connecting this back to verifying hoare triples we can define a hoare triple and a query over all its variables

$$\{P\}\ s\ \{Q\} \quad p(\overline{x})$$

In addition we say that

$$\mathtt{wp}(s, p(\overline{x})) = q(\overline{e})$$

and we have $\mathbb{C}$ as the set of horn clauses generated by wp. Then we can say that the following set of horn clauses

$$\mathbb{C} \cup \{P \to q(\overline{e}),\ p(\overline{x}) \to Q\}$$

is satisfiable if and only if the hoare triple $\{P\}\ s\ \{Q\}$ is valid.

4.8 Solving Horn Clauses

4.9 Strongest Postcondition

First Let's define an operator post which computes the strongest postcondition of of a formula φ w.r.t a set of variables $\overline{x}$. If we define $\overline{y}$ as the set of variables in φ that are not in $\overline{x}$, i.e. $\overline{y} = \text{Free}(\varphi) \setminus \overline{x}$, then we can define the strongest postcondition as follows

$$\texttt{post}(\varphi, \overline{x}) \triangleq \exists \overline{y}.\ \varphi$$

The following property holds for $\overline{\varphi}\ \overline{x}$

$$\texttt{post}(\varphi_1 \wedge \varphi_2 \wedge \ldots, \overline{x}) \equiv \texttt{post}(\varphi_1, \overline{x}) \wedge \texttt{post}(\varphi_2, \overline{x}) \wedge \ldots$$

Example 4.8. Using strongest postcondition
Consider the formula

$$\texttt{post}(y = x + 1 \wedge x \geqslant 0, y)$$

Here we have $\overline{x} = \{y\}$ and $\overline{y} = \{x\}$. Thus applying the definition we have

$$\begin{aligned}
\texttt{post}(y = x + 1 \wedge x \geqslant 0, y) &\equiv \exists x.\ (y = x + 1 \wedge x \geqslant 0) \\
&\equiv \exists x.\ (x = y - 1 \wedge x \geqslant 0) \\
&\equiv y - 1 \geqslant 0 \\
&\equiv y \geqslant 1
\end{aligned}$$

Another example

$$\texttt{post}(y = x + 1 \wedge (x = 0 \vee x = 1), y)$$

Here we have $\overline{x} = \{y\}$ and $\overline{y} = \{x\}$. Thus applying the definition we have

$$\begin{aligned}
\texttt{post}(y = x + 1 \wedge (x = 0 \vee x = 1), y) &\equiv \exists x.\ (y = x + 1 \wedge (x = 0 \vee x = 1)) \\
&\equiv \exists x.\ (x = y - 1 \wedge (x = 0 \vee x = 1)) \\
&\equiv (y - 1 = 0) \vee (y - 1 = 1) \\
&\equiv (y = 1) \vee (y = 2)
\end{aligned}$$

4.10 The general approach

1. Initialize - We start with a solution Σ which maps each query p to $\bot$, i.e. $\Sigma(p) \equiv \bot$ for all queries p.
2. Iteration - We pick any clause whos head is a query, i.e. a clause in the form

   $$p(\overline{x}) \wedge \varphi \to q(\overline{y})$$

   and then compute

   $$\psi \triangleq \texttt{post}(\Sigma(p(\overline{x})) \wedge \varphi, \overline{y})$$

   Subsumption - If $\Sigma(q) \equiv \psi'$, we then update our solution as follows

   $$\Sigma(q) \triangleq \psi' \vee \psi$$

Example 4.9. Consider the following set of horn clauses

$$x = 0 \to q(x) \tag{1}$$
$$(q(y) \wedge y < 6 \wedge x = y + 1) \to q(x) \tag{2}$$

1. Initialization - We start with the solution

   $$\Sigma(q) \equiv \bot$$

2. Iteration 1 - We pick clause 1 and compute

$$\psi \triangleq \mathsf{post}(\Sigma(\bot) \wedge (x = 0), x) \equiv \mathsf{post}(x = 0, x)$$

Here we have $\overline{x} = \{x\}$ and $\overline{y} = \emptyset$. Thus applying the definition we have

$$\mathsf{post}(x = 0, x) \equiv \exists \emptyset.\ (x = 0) \equiv x = 0$$

Now we update our solution

$$\Sigma(q) \triangleq \bot \vee (x = 0) \equiv x = 0$$

3. Iteration 2 - We pick clause 2 and compute

$$\psi \triangleq \mathsf{post}(\Sigma(q(y)) \wedge (y < 6 \wedge x = y + 1), x)$$

Here we have $\overline{x} = \{x\}$ and $\overline{y} = \{y\}$. Thus applying the definition we have

$$\begin{aligned}\mathsf{post}((y = 0) \wedge (y < 6 \wedge x = y + 1), x) &\equiv \exists y.\ (y = 0 \wedge y < 6 \wedge x = y + 1)\\ &\equiv \exists y.\ (y = 0 \wedge x = y + 1)\\ &\equiv x = 1\end{aligned}$$

Now we update our solution

$$\Sigma(q) \triangleq (x = 0) \vee (x = 1)$$

4. Iteration 3 - We pick clause 2 again and compute

$$\psi \triangleq \mathsf{post}(\Sigma(q(y)) \wedge (y < 6 \wedge x = y + 1), x)$$

Here we have $\overline{x} = \{x\}$ and $\overline{y} = \{y\}$. Thus applying the definition we have

$$\begin{aligned}\mathsf{post}(((y = 0) \vee (y = 1)) \wedge (y < 6 \wedge x = y + 1), x) &\equiv \exists y.\ (((y = 0) \vee (y = 1)) \wedge y < 6 \wedge x = y + 1)\\ &\equiv \exists y.\ ((y = 0 \wedge x = y + 1) \vee (y = 1 \wedge x = y + 1))\\ &\equiv (x = 1) \vee (x = 2)\end{aligned}$$

Now we update our solution

$$\Sigma(q) \triangleq (x = 0) \vee (x = 1) \vee (x = 2)$$

5. Further Iterations - Continuing this process we eventually arrive at

$$\Sigma(q) \equiv (x = 0) \vee (x = 1) \vee (x = 2) \vee (x = 3) \vee (x = 4) \vee (x = 5) \vee (x = 6)$$

6. Termination - At this point further iterations will not yield any new information as applying clause 2 again will only produce values of x greater than 6 which do not satisfy the $y < 6$ constraint in the body of the clause. Thus we have reached a fixed point and our final solution is

$$\Sigma(q) \equiv (x = 0) \vee (x = 1) \vee (x = 2) \vee (x = 3) \vee (x = 4) \vee (x = 5) \vee (x = 6)$$

4.11 Addressing divergence

In the case where a set of clauses have solutions represented by infinite states the naive approach will diverge. As a motivating example Let's consider the following set of horn clauses

$$x = 1 \wedge n \geqslant 1 \rightarrow q(x, n) \tag{1}$$

$$q(y, n) \wedge x = y + 1 \wedge y < n \rightarrow q(x, n) \tag{2}$$

Applying the naive approach we would end up with the solution

$$\Sigma(q) \equiv (x = 1) \vee (x = 2) \vee (x = 3) \vee \ldots$$

Intuitively speaking here the clauses can be understood as

1. Clause 1 - The base case where x is initialized to 1 for any $n \geqslant 1$.
2. Clause 2 - The recursive case where x is incremented by 1 as long as it is less than n.

As a while loop equivalently

$$\{n \geqslant 1\} \texttt{ while } x < n \texttt{ do } x := x + 1 \{x \geqslant 1\}$$

The predicate $q(x, n)$ here represents the set of *reachable states* (x, n) during the execution of the loop. We start with the pair $(x = 1, n \geqslant 1)$ then get the following graph

$$(1, n) \to (2, n) \to (3, n) \to \ldots \to (n, n)$$

Naturally for an arbitrary n this set is infinite as x can take any value from 1 to n. Thus the naive approach attempts to enumerate all these states leading to divergence.

Evidently this solution diverges as there is no finite representation of all the possible values of x and n. The crux of the issue is that the naive approach preserves too much information about the individual states. Whereas for verification purposes its sufficient to reason about the abstract property describing the set of states. Let's consider the terminating condition

$$q(x, n) \wedge x \geqslant n \to x = n \wedge x \geqslant 0$$

This represents the loops post condition, i.e. when the loop terminates x must equal n and be non-negative. An important observation to make here is that we don't need to know every state (x, n) that q can take, we only need to know the two invariants that hold for all reachable states, namely

$$x \geqslant 0 \wedge x \leqslant n$$

Another way of viewing this is that we are simply weakening our granular predicates to coarser ones that capture the essential properties we care about. Intuitively this gives rise to the general notion of *abstraction* where we abstract away unnecessary details to focus on the relevant properties. In our example we represent the set of reachable states $q(x, n)$ using the abstract property $x \geqslant 0 \wedge x \leqslant n$ rather than enumerating all possible pairs (x, n).

4.11.1 Predicate abstraction

In predicate abstraction as opposed to representing a set of states by all possible concrete states we represent them using predicates that capture the essential properties of those states. Formally we define a set of predicates P as follows

$$P \triangleq \{p_1(\overline{x}), p_2(\overline{x}), \ldots, p_k(\overline{x})\}$$

where each p_i is a formula in the theory T over the variables $\overline{x}$.

Example 4.10. Consider the following predicate set

$$P \triangleq \{x \geqslant 0, x \geqslant 1, y \geqslant -1\}$$

We can see here that each element of the set is a boolean-valued formula over the variables x and y. Each program state becomes a bitvector (vector of booleans) representing which predicates hold true in that state. For example Let's consider

$$\sigma \triangleq \{x \mapsto 2, y \mapsto 0\}$$

In this state we have

$$x \geqslant 0 \equiv \top$$
$$x \geqslant 1 \equiv \top$$
$$y \geqslant -1 \equiv \top$$

Thus the abstract representation of this state is

$$\langle \top, \top, \top \rangle$$

We say that the k predicates in P induce an abstract domain of 2^k abstract states, in other words they define a finite vocabulary for describing program states. The key idea is that instead of tracking exact values of variables we track which predicates hold true in a given state. This allows us to represent potentially infinite sets of concrete states using a finite set of abstract states.

We can define an abstraction function

$$\alpha : \texttt{States} \times \texttt{Predicates} \to \texttt{AbstractStates}$$

which maps concrete program states to abstract states based on the predicates in P. Given a concrete state σ we define

$$\alpha(\varphi, P) \triangleq \bigwedge_{p_i \in P} \varphi \to p_i$$

in plain english the idea says that we can abstract a concrete formula φ (potentially describing a set of a reachable states via constraints) and a set of assoicated atomic predicates P into an abstract state by conjoining all predicates in P that are implied by φ.

Example 4.11. Let's consider

$$\varphi \triangleq (x = 5 \wedge y = 0) \quad \text{and} \quad P \triangleq \{x \geqslant 0, x \geqslant 1, y \geqslant -1\}$$

Checking which predicates are implied by φ we have

$$\begin{aligned}(x = 5 \wedge y = 0) &\to (x \geqslant 0) \equiv \top \\ (x = 5 \wedge y = 0) &\to (x \geqslant 1) \equiv \top \\ (x = 5 \wedge y = 0) &\to (y \geqslant -1) \equiv \top\end{aligned}$$

Thus we have

$$\alpha(\varphi, P) \equiv (x \geqslant 0) \wedge (x \geqslant 1) \wedge (y \geqslant -1)$$

This represents our *strongest conjunction* of predicates from P that are implied by φ.

Example 4.12. Let's consider

$$\varphi \triangleq (x \geqslant 1 \wedge y = -2) \quad \text{and} \quad P \triangleq \{x \geqslant 0, x \geqslant 1, y \geqslant -1\}$$

Checking which predicates are implied by φ we have

$$\begin{aligned}(x \geqslant 1 \wedge y = -2) &\to (x \geqslant 0) \equiv \top \\ (x \geqslant 1 \wedge y = -2) &\to (x \geqslant 1) \equiv \top \\ (x \geqslant 1 \wedge y = -2) &\to (y \geqslant -1) \equiv \bot\end{aligned}$$

Thus we have

$$\alpha(\varphi, P) \equiv (x \geqslant 0) \wedge (x \geqslant 1)$$

We can see here that $y \geqslant -1$ is violated by the choice of φ thus it is not included in the abstraction.

Importantly we understand that $\alpha(\varphi, P)$ represents the *strongest conjunction of predicates* from P that are implied by φ. This means that any other conjunction of predicates from P that is implied by φ must also imply $\alpha(\varphi, P)$. In other words $\alpha(\varphi, P)$ is the most precise abstraction of φ using the predicates in P.

In an intuitive sense we can think of $\alpha(\varphi, P)$ as the best possible summary of the information in φ that can be expressed using the predicates in P while still ensuring termination.

4.11.2 Abstract Strongest Postcondition

We can define a new operator `post#` which similarly to `post` computes the strongest postcondition of a formula φ w.r.t a set of variables $\overline{x}$ but now also incorporates predicate abstraction using a set of predicates P. Formally we define

$$\texttt{post\#}(\varphi, \overline{x}, P) \triangleq \alpha(\texttt{post}(\varphi, \overline{x}), P)$$

Example 4.13. Let's consider our predicate set from before

$$P \triangleq \{x \geqslant 0, x \geqslant 1, y \geqslant -1\}$$

now let's compute the following abstract strongest postconditions

$$\begin{aligned} \texttt{post\#}(x = y + 1 \wedge y \geqslant -1, x) &\equiv \alpha(\texttt{post}(x = y + 1 \wedge y \geqslant -1, x), P) \\ &\equiv \alpha(\exists y.\ (x = y + 1 \wedge y \geqslant -1), P) \\ &\equiv \alpha(x \geqslant 0, P) \end{aligned}$$

$$\begin{aligned} \texttt{post\#}(x = y + 1 \wedge y = 0, x) &\equiv \alpha(\texttt{post}(x = y + 1 \wedge y = 0, x), P) \\ &\equiv \alpha(\exists y.\ (x = y + 1 \wedge y = 0), P) \\ &\equiv \alpha(x = 1, P) \\ &\equiv x \geqslant 0 \wedge x \geqslant 1 \end{aligned}$$

$$\begin{aligned} \texttt{post\#}(x = y + 1 \wedge y = 1, x) &\equiv \alpha(\texttt{post}(x = y + 1 \wedge y = 1, x), P) \\ &\equiv \alpha(\exists y.\ (x = y + 1 \wedge y = 1), P) \\ &\equiv \alpha(x = 2, P) \\ &\equiv x \geqslant 0 \wedge x \geqslant 1 \end{aligned}$$

4.11.3 General approach with abstraction

We can now modify our general approach to incorporate predicate abstraction as follows

1. Initialize - We start with a solution Σ which maps each query p to $\bot$, i.e. $\Sigma(p) \equiv \bot$ for all queries p.
2. Iteration - We pick any clause whos head is a query, i.e. a clause in the form

   $$p(\overline{x}) \wedge \varphi \to q(\overline{y})$$

   and then compute

   $$\psi \triangleq \texttt{post\#}(\Sigma(p(\overline{x})) \wedge \varphi, \overline{y}, P) \equiv \alpha(\texttt{post}(\Sigma(p(\overline{x})) \wedge \varphi, \overline{y}), P)$$

   Subsumption - If $\Sigma(q) \equiv \psi'$, we then update our solution as follows

   $$\Sigma(q) \triangleq \psi' \vee \psi$$

Example 4.14. Once again Let's consider our while loop

$$\{n \geqslant 1\}\ \texttt{while}\, x < n\ \texttt{do}\, x := x + 1\ \{x = n\}$$

and the corresponding horn clauses

$$x = 1 \wedge n \geqslant 1 \to q(x, n) \tag{1}$$

$$q(y, n) \wedge x = y + 1 \wedge y < n \to q(x, n) \tag{2}$$

$$q(x, n) \wedge x \geqslant n \to x = n \tag{3}$$

Now Let's choose the predicate set

$$P \triangleq \{x \geqslant 1, x \geqslant 0, y \geqslant -1, x \leqslant n\}$$

1. Initialization - We start with the solution

   $$\Sigma(q) \equiv \bot$$

2. Iteration 1 - We pick clause 1 and compute

   $$\psi \triangleq \texttt{post\#}(\Sigma(\bot) \wedge (x = 1 \wedge n \geqslant 1), (x, n), P) \equiv \alpha(\texttt{post}(x = 1 \wedge n \geqslant 1, (x, n)), P)$$

Here we have $\overline{x} = \{x, n\}$ and $\overline{y} = \emptyset$. Thus applying the definition we have

$$\begin{aligned} \texttt{post}(x = 1 \land n \geqslant 1, (x, n)) &\equiv \exists \emptyset.\ (x = 1 \land n \geqslant 1) \\ &\equiv x = 1 \land n \geqslant 1 \end{aligned}$$

Now applying the abstraction function we have

$$\begin{aligned} \alpha(x = 1 \land n \geqslant 1, P) &\equiv \bigwedge_{p_i \in P} (x = 1 \land n \geqslant 1) \to p_i \\ &\equiv (x \geqslant 1) \land (x \geqslant 0) \land (y \geqslant -1) \land (x \leqslant n) \end{aligned}$$

Now we update our solution

$$\Sigma(q) \equiv (x \geqslant 1) \land (x \geqslant 0) \land (y \geqslant -1) \land (x \leqslant n)$$

3. Iteration 2 - We pick clause 2 and compute

   $\psi \triangleq \texttt{post\#}(\Sigma(q(y, n)) \land (y < n \land x = y+1), (x, n), P) \equiv \alpha(\texttt{post}(\Sigma(q(y, n)) \land (y < n \land x = y+1), (x, n)), P)$

   Here we have $\overline{x} = \{x, n\}$ and $\overline{y} = \{y\}$. Thus applying the definition we have

$$\begin{aligned} \texttt{post}(q(y, n) \land (y < n \land x = y + 1), (x, n)) &\equiv \exists y.\ (q(y, n) \land (y < n \land x = y + 1)) && (1) \\ &\equiv \exists y.\ ((y \geqslant 1) \land (y < n) \land (x = y + 1)) && (2) \\ &\equiv (x \geqslant 2) \land (x \leqslant n) && (3) \end{aligned}$$

   If its not clear how 2. to 3. follows we can break it down as follows

$$\begin{aligned} \exists y.\ ((y \geqslant 1) \land (y < n) \land (x = y + 1)) &\equiv \exists y.\ ((y \geqslant 1) \land (y < n) \land (y = x - 1)) \\ &\equiv (x - 1 \geqslant 1) \land (x - 1 < n) && \text{(Substituting } y\text{)} \\ &\equiv (x \geqslant 2) \land (x \leqslant n) \end{aligned}$$

   Now applying the abstraction function we have

$$\begin{aligned} \alpha((x \geqslant 2) \land (x \leqslant n), P) &\equiv \bigwedge_{p_i \in P} ((x \geqslant 2) \land (x \leqslant n)) \to p_i \\ &\equiv (x \geqslant 1) \land (x \geqslant 0) \land (x \leqslant n) \end{aligned}$$

   Now we update our solution

$$\Sigma(q) \equiv \bigwedge_{p_i \in P} p_i \lor ((x \geqslant 1) \land (x \geqslant 0) \land (x \leqslant n)) \equiv \bigwedge_{p_i \in P} p_i$$

4. Further Iterations - Continuing this process we see that further iterations will not yield any new information as applying clause 2 again will only produce values of x greater than n which do not satisfy the $y < n$ constraint in the body of the clause. Thus we have reached a fixed point and our final solution is

$$\Sigma(q) \equiv \bigwedge_{p_i \in P} p_i$$

Importantly we should then ask ourselves if this solution is sufficient to prove the validity of our original hoare triple. Recalling the termination clause

$$q(x, n) \land x \geqslant n \to x = n$$

we can see that substituting our solution for q we have

$$((x \geqslant 1) \land (x \geqslant 0) \land (y \geqslant -1) \land (x \leqslant n)) \land x \geqslant n \to x = n$$

Simplifying the left hand side we have

$$(x \geqslant n) \land (x \leqslant n) \to x = n$$

which is valid. Thus, we have successfully verified the hoare triple using predicate abstraction to ensure termination. From this can say that $\Sigma \models \mathbb{C}$ where $\mathbb{C}$ is our set of horn clauses.

For the sake of completeness Let's examine some other possible choices of predicate sets P.

- Choice 1 - $P \triangleq \{x \geqslant 0, x \geqslant 1, x \leqslant n\}$
  Clearly this choice will also work as it captures the essential invariants needed to prove the hoare triple.
- Choice 2 - $P \triangleq \{x \geqslant 1, x \leqslant n\}$
  This one also works as the condition $x \geqslant 1$ is sufficient to imply $x \geqslant 0$.
- Choice 3 - $P \triangleq \{y \geqslant -1, x \leqslant n\}$
  This one fails as we lose information about the lower bound of x.

Example 4.15. Computing a solution
Consider the following set of horn clauses

$$\begin{aligned}
&(1) \quad x = 0 \wedge y = 0 \rightarrow q(x, y, n, f)\\
&(2) \quad q(x', y', n, f) \wedge \underbrace{x = x' + 1}_{s_1} \wedge \underbrace{f = 1 \wedge y = y' + 1}_{b_1} \wedge \underbrace{x' < n}_{l} \rightarrow q(x, y, n, f)\\
&(3) \quad q(x', y', n, f) \wedge \underbrace{x = x' + 1}_{s_1} \wedge \underbrace{f \neq 1 \wedge y = y' - 1}_{b_2} \wedge \underbrace{x' < n}_{l} \rightarrow q(x, y, n, f)\\
&(4) \quad q(x, y, n, f) \wedge f = 1 \rightarrow y \geqslant 0
\end{aligned}$$

Similar to before we can interpret these clauses as representing a program with a loop and two branches depending on the value of f. So expressed as a program we have

```
x := 0; y := 0;
while (x < n) {
    x := x + 1;
    if (f == 1) {
        y := y + 1;
    } else {
        y := y - 1;
    }
}
assert (f == 1) -> (y >= 0);
```

Let's choose the predicate set

$$P \triangleq \{y \geqslant 0, f \neq 1\}$$

Applying our abstract strongest postcondition approach we have the following iterations

1. Initialization - We start with the solution
   $$\Sigma(q) \equiv \bot$$
2. Iteration 1 - We pick clause 1 and compute
   $$\psi \triangleq \mathtt{post\#}(\Sigma(\bot) \wedge (x = 0 \wedge y = 0), (x, y, n, f), P) \equiv \alpha(\mathtt{post}(x = 0 \wedge y = 0, (x, y, n, f)), P)$$
   Here we have $\overline{x} = \{x, y, n, f\}$ and $\overline{y} = \emptyset$. Thus applying the definition we have
   $$\begin{aligned}
   \mathtt{post}(x = 0 \wedge y = 0, (x, y, n, f)) &\equiv \exists \emptyset.\ (x = 0 \wedge y = 0)\\
   &\equiv x = 0 \wedge y = 0
   \end{aligned}$$
   Now applying the abstraction function we have
   $$\begin{aligned}
   \alpha(x = 0 \wedge y = 0, P) &\equiv \bigwedge_{p_i \in P} (x = 0 \wedge y = 0) \rightarrow p_i\\
   &\equiv (y \geqslant 0)
   \end{aligned}$$
   Now we update our solution
   $$\Sigma(q) \equiv (y \geqslant 0)$$

3. Iteration 2 - We pick clause 2 and compute

$$\begin{aligned}\psi &\triangleq \texttt{post\#}(\Sigma(q(x', y', n, f)) \wedge (s_1 \wedge b_1 \wedge l), (x, y, n, f), P) \\ &\equiv \alpha(\texttt{post}(\Sigma(q(x', y', n, f)) \wedge (s_1 \wedge b_1 \wedge l), (x, y, n, f)), P)\end{aligned}$$

Here we have $\overline{x} = \{x, y, n, f\}$ and $\overline{y} = \{x', y'\}$. Thus applying the definition we have

$$\begin{aligned}\texttt{post}((y' \geqslant 0) \wedge (s_1 \wedge b_1 \wedge l)), (x, y, n, f) &\equiv \exists x', y'.\ ((y' \geqslant 0) \wedge (s_1 \wedge b_1 \wedge l)) \\ &\equiv \exists x', y'.\ (y' \geqslant 0 \wedge (x = x' + 1) \wedge f = 1 \wedge y = y' + 1 \wedge x' < n) \\ &\equiv (y - 1 \geqslant 0) \wedge (f = 1) \wedge (x - 1 < n) \\ &\equiv (y \geqslant 1) \wedge (f = 1) \wedge (x < n + 1)\end{aligned}$$

Now applying the abstraction function we have

$$\begin{aligned}\alpha((y \geqslant 1) \wedge (f = 1) \wedge (x < n + 1), P) &\equiv \bigwedge_{p_i \in P} ((y \geqslant 1) \wedge (f = 1) \wedge (x < n + 1)) \rightarrow p_i \\ &\equiv (y \geqslant 0)\end{aligned}$$

Now we update our solution

$$\Sigma(q) \equiv (y \geqslant 0) \vee (y \geqslant 0) \equiv (y \geqslant 0)$$

4. Iteration 3 - We pick clause 3 and compute

$$\begin{aligned}\psi &\triangleq \texttt{post\#}(\Sigma(q(x', y', n, f)) \wedge (s_1 \wedge b_2 \wedge l), (x, y, n, f), P) \\ &\equiv \alpha(\texttt{post}(\Sigma(q(x', y', n, f)) \wedge (s_1 \wedge b_2 \wedge l), (x, y, n, f)), P)\end{aligned}$$

Here we have $\overline{x} = \{x, y, n, f\}$ and $\overline{y} = \{x', y'\}$. Thus applying the definition we have

$$\begin{aligned}\texttt{post}((y' \geqslant 0) \wedge (s_1 \wedge b_2 \wedge l)), (x, y, n, f) &\equiv \exists x', y'.\ ((y' \geqslant 0) \wedge (s_1 \wedge b_2 \wedge l)) \\ &\equiv \exists x', y'.\ (y' \geqslant 0 \wedge (x = x' + 1) \wedge f \neq 1 \wedge y = y' - 1 \wedge x' < n) \\ &\equiv (y + 1 \geqslant 0) \wedge (f \neq 1) \wedge (x - 1 < n) \\ &\equiv (y \geqslant -1) \wedge (f \neq 1) \wedge (x < n + 1)\end{aligned}$$

Now applying the abstraction function we have

$$\begin{aligned}\alpha((y \geqslant -1) \wedge (f \neq 1) \wedge (x < n + 1), P) &\equiv \bigwedge_{p_i \in P} ((y \geqslant -1) \wedge (f \neq 1) \wedge (x < n + 1)) \rightarrow p_i \\ &\equiv (f \neq 1)\end{aligned}$$

Now we update our solution

$$\Sigma(q) \equiv (y \geqslant 0) \vee (f \neq 1)$$

5. Termination - At this point further iterations will not yield any new information as applying clauses 2 and 3 again will only produce values of x greater than n which do not satisfy the $x' < n$ constraint in the body of the clauses. Thus we have reached a fixed point and our final solution is

$$\Sigma(q) \equiv (y \geqslant 0) \vee (f \neq 1)$$

6. Verification - Finally we check if our solution is sufficient to prove the validity of our original assertion clause 4. Substituting our solution for q we have

$$((y \geqslant 0) \vee (f \neq 1)) \wedge f = 1 \rightarrow y \geqslant 0$$

Simplifying the left hand side we have

$$(y \geqslant 0) \rightarrow y \geqslant 0$$

which is valid. Thus, we have successfully verified the assertion using predicate abstraction to ensure termination. From this can say that $\Sigma \models \mathbb{C}$ where $\mathbb{C}$ is our set of horn clauses.

4.12 Correctness

A good question to ask at this point is why exactly is it correct to use an abstract predicate if it inherently loses information about the concrete states? When we solve horn clauses or compute reachable states we can often have recurrences like

$$\Sigma_{i+1} = \texttt{post}(\Sigma_i)$$

where post is our postcondition operator which computes the precise concrete states that can be reached from a given set of states after one step of execution. But as we already discussed this process can diverge if the set of reachable states is infinite. Instead we use an abstract postcondition operator post# defined as

$$\texttt{post\#}(\varphi) \triangleq \alpha(\texttt{post}(\varphi), P)$$

which represents an over-approximation of the concrete post. If we introduce a *concretization* function γ which maps abstract states back to concrete states we can have that

$$\texttt{post}(\varphi) \subseteq \gamma(\texttt{post\#}(\varphi)) \tag{12}$$

An important understanding here being that replacing post with post# *does not* make verification unsound, i.e. it can never lead to us proving a false property. It does however lose precision as we are now reasoning about an over-approximation of the reachable states.

Stated more formally lets consider the following

$$\begin{aligned} \Sigma_0 &= \text{initial states} \\ \Sigma_{i+1} &= \texttt{post}(\Sigma_i) \end{aligned}$$

Let's say that after k iterations we reach a fixed point Σ_k such that

$$\Sigma_k = \texttt{post}(\Sigma_k)$$

The central idea of the abstract interpretation framework is that if we instead define

$$\begin{aligned} \Sigma'_0 &= \text{initial states} \\ \Sigma'_{i+1} &= \texttt{post\#}(\Sigma'_i) \end{aligned}$$

and after m iterations reach a fixed point Σ'_m such that

$$\Sigma'_m = \texttt{post\#}(\Sigma'_m)$$

then we have that

$$\Sigma_k \subseteq \gamma(\Sigma'_m)$$

In other words the concrete reachable states Σ_k are always contained within the concretization of the abstract reachable states Σ'_m. This ensures that any property proven about the abstract states also holds for the concrete states, thus maintaining soundness in verification.

4.12.1 Trusting abstraction

Lets consider the clause

$$b \triangleq q \rightarrow \neg\psi$$

This clause represents the saftey property namely

$$\text{If } q \text{ holds then } \psi \text{ must not hold}$$

or more plainly just: "No reachable state should satisfy the error condition ψ". So ψ or $\psi_{\texttt{error}}$ represents the error states we want to avoid. As before we informally have the following relation

$$\underbrace{\Sigma_0 \subseteq \Sigma_1 \ldots \subseteq \Sigma_k}_{\text{Concrete states}} \subseteq \ldots \subseteq \Sigma\# \quad \text{and} \quad \psi_{\texttt{error}}$$

Theres two situations we can now consider, namely

- $\Sigma^\# \not\subseteq \psi$ - In this case we can say that our program is safe as none of the reachable states satisfy the error condition. Thus we have verified our program.
- $\Sigma^\# \subseteq \psi$ - In this case we have two possibilities
  - False positive - Here the error condition is only satisfied in the abstract states but not in any concrete reachable states. This can happen due to the over-approximation introduced by abstraction. In this case we cannot conclude that the program is unsafe and may need to refine our abstraction or use other techniques to rule out the false positive.
  - True positive - Here the error condition is satisfied in some concrete reachable states. In this case we can conclude that the program is indeed unsafe and needs to be fixed.

The important takeaway here is that *abstract verification can only prove safety*, i.e. if no abstract reachable states satisfy the error condition then the program is safe. However if some abstract reachable states do satisfy the error condition we cannot immediately conclude that the program is unsafe due to the possibility of false positives introduced by abstraction. In short we would say that abstract verification is **sound** but **not complete**.

4.13 Complete Lattice

A lattice L is a mathematical structure $(L, \sqsubseteq)$ where

- L is a set of elements. In our case these are formulas built from the predicates in P.
- $\sqsubseteq$ is a *partial order* relation on L, i.e. for any $a, b, c \in L$ we have
  - Reflexivity: $a \sqsubseteq a$
  - Antisymmetry: If $a \sqsubseteq b$ and $b \sqsubseteq a$ then $a = b$
  - Transitivity: If $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$
- For any two elements $a, b \in L$ there exists a *least upper bound* (join/union/or) denoted $a \vee b$ such that
  - $a \sqsubseteq (a \vee b)$ and $b \sqsubseteq (a \vee b)$
  - For any $c \in L$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$ we have $(a \vee b) \sqsubseteq c$
- For any two elements $a, b \in L$ there exists a *greatest lower bound* (meet/intersection/and) denoted $a \wedge b$ such that
  - $(a \wedge b) \sqsubseteq a$ and $(a \wedge b) \sqsubseteq b$
  - For any $c \in L$ such that $c \sqsubseteq a$ and $c \sqsubseteq b$ we have $c \sqsubseteq (a \wedge b)$

We say that a lattice is *complete* if for any subset $S \subseteq L$ there exists a least upper bound $\bigvee S$ and a greatest lower bound $\bigwedge S$ in L.

Applying this to the case of logic we establish that

$$a \sqsubseteq b \iff a \to b$$

The intuition of this is that a describes a smaller (more precise) set of states than b. Whereas b is more general (less precise) than a. Thus if a implies b then any state satisfying a must also satisfy b.

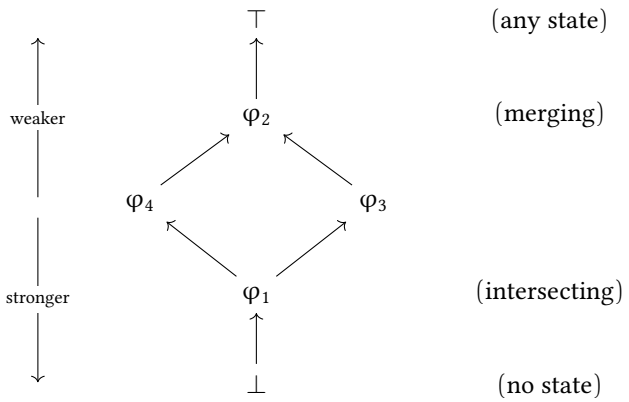
Example 4.16. Lets consider

$$P \triangleq \{f \neq 1, y \geqslant 0\}$$

Now lets we consider the formulas

$$\varphi_1 \triangleq (f \neq 1) \wedge (y \geqslant 0)$$
$$\varphi_2 \triangleq (f \neq 1) \vee (y \geqslant 0)$$
$$\varphi_3 \triangleq (y \geqslant 0)$$
$$\varphi_4 \triangleq (f \neq 1)$$

If we draw the Hasse diagram of these formulas under the implication ordering we have



Here we can see that the top element $\top$ represents the least precise formula which is implied by all other formulas. Conversely the bottom element $\bot$ represents the most precise formula which implies all other formulas. We can also see that for any two formulas we can compute their least upper bound (join) and greatest lower bound (meet) within this lattice.
The importance of this structure is that because the set of formulas under implication ordering forms a complete lattice, we can apply fixed point theorems such as Kleene's fixed point theorem to guarantee the existence of fixed points when using monotonic functions like our abstract strongest postcondition operator `post#`. This ensures that our iterative approach to solving horn clauses will eventually converge to a fixed point, providing a sound basis for verification using predicate abstraction.

4.13.1 Kleene's Fixed Point Theorem

The essential property of our abstract strongest postcondition operator `post#` is that it is *monotonic* w.r.t the implication ordering. Formally this means that for any two formulas φ_1 and φ_2 such that

$$\varphi_1 \sqsubseteq \varphi_2 \rightarrow \texttt{post\#}(\varphi_1) \sqsubseteq \texttt{post\#}(\varphi_2)$$

This property intuitively implies that during our analysis applying `post#` will never lead to us moving to a less precise (weaker) formula if we start from a more precise (stronger) formula. That is, we never move downwards in the lattice when applying `post#`. Abstractly we can denote the analysis as always abiding by the following relation

$$\bot \sqsubseteq \texttt{post\#}(\bot) \sqsubseteq \texttt{post\#}(\texttt{post\#}(\bot)) \sqsubseteq \ldots \sqsubseteq \top$$

Furthermore because the lattice has a finite hight - due to the finite number of predicates in P - we are guaranteed to eventually reach a fixed point φ^* such that

$$\varphi^* = \texttt{post\#}(\varphi^*)$$

This is a direct consequence of Kleene's fixed point theorem which states that any monotonic function on a complete lattice has a least fixed point which can be reached by iteratively applying the function starting from the least element of the lattice (in our case $\bot$). Formally

Theorem 1 (Kleene's Fixed Point Theorem)**.** Let $(L, \sqsubseteq)$ be a complete lattice and let $f : L \rightarrow L$ be a monotonic function. Then the least fixed point of f, denoted $\mathrm{lfp}(f)$, can be computed as

$$\mathrm{lfp}(f) = \bigvee_{n \geqslant 0} f^n(\bot)$$

where f^n denotes the n-th iteration of f starting from the least element $\bot$ of the lattice.

4.14 Addressing False Positives in $\Sigma\#$

One challenge that we noted earlier with using predicate abstraction is the possibility of false positives. These occur when the abstract analysis indicates that an error state is reachable, but in reality, no concrete execution can reach that state. This can happen due to the over-approximation introduced by abstraction.

An important question to ask here is what can we do if an abstract solution $\Sigma\#$ suggests that our program is unsafe but it might just be missing information. Here we can introduce the notion of *abstraction refinement*.

4.14.1 Abstraction Refinement

At a high level our goal is to add new relevant predicates to our abstraction to make it more precise - just enough to eliminate the false positives. We can describe this process as follows

1. Define the 'bound' clause - We start by defining a clause that captures the error condition we want to avoid. This clause can be expressed as
$$b \triangleq q \rightarrow \neg \psi_{error}$$
where q is the query representing the reachable states and ψ_{error} is the error condition.

2. Check for violation - We then check if our current abstract solution $\Sigma\#$ satisfies the bound clause b. If it does not, i.e. if $\Sigma\# \subseteq \psi_{error}$ or equivalently $\Sigma\# \not\models b$, then we have a potential violation.

3. Analyze counterexample - If a violation is detected, we analyze the counterexample provided by the abstract solution. This involves examining the abstract states that lead to the error condition and identifying which predicates are missing or insufficiently precise.

4. Compute new predicates - We compute a new formula φ_{new} such that
$$\Sigma(q) \rightarrow \varphi_{new} \quad \text{and} \quad \varphi_{new} \wedge \psi_{error} \equiv \bot$$
Here the formula φ_{new} is also known as the *Craig interpolant.* As the name implies it interpolates - sits between - the concrete solution $\Sigma(q)$ and the error condition ψ_{error}.

5. Refine predicate set - We then refine our predicate set P by adding new predicates extracted from φ_{new}. This enhances the precision of our abstraction.