Chapter 4
Verification Using Automated Theorem Provers

An automated theorem prover is a tool that determines, automatically or semi-automatically, the validity of formulas in a particular logic. Automated theorem provers have been put to use in many applications. They have been used to solve open problems in mathematics, such as Robbin's problem in boolean algebra [140], which was open since the 1930s, and various open problems about quasi-groups [188]. They have also been used to prove interesting properties about real-world systems, properties that would have been hard, difficult or tedious to prove by hand. For example, automated theorem provers have been used to verify micro-processors [19, 20, 164], communication protocols [20], concurrent algorithms [20], and various properties of software systems [9, 58, 100, 130, 132, 164, 168].

From the perspective of this book, we are interested in this later use of automated theorem provers: verification of systems. Within this space, there are two broad categories of automated theorem provers. First, there are *interactive theorem provers*. Examples of such theorem provers include Coq [13], NuPrl [36], ACL2 [20], and Twelf [168]. Such theorem provers are essentially proof assistants that allow the programmer to state and prove theorems interactively. The mode and level of human interaction varies by theorem prover. For example, in the Coq theorem prover, the proof is built up in full detail by the programmer using a scripting language. In ACL2, the programmer provides hints in the form of annotations and in the form of helper lemmas that the theorem prover should attempt to prove first. Such interactive theorem provers have been used to establish the correctness of software systems, some notable examples including an entire compiler and an entire database management system. Such proof efforts require an inordinate amount of manual work, since they require formalizing every last detail in full formal logic. However, precisely because of this level of detail, once the verification is done, it provides a very high-level of assurance.

At the other end of the automation spectrum, one finds *fully automated theorem provers*, which are the theorem provers we will be interested in for this book. Such theorem provers simply take a formula, typically in some restricted logic, and return either *valid* or *invalid*. If *valid* is returned, then we know that the original formula is a tautology. If *invalid* is returned, then we learn different things depending on the completeness of the theorem prover: if the theorem prover is complete, we learn that the formula is *not* a tautology; otherwise we learn nothing – the

S. Kundu et al., *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, DOI 10.1007/978-1-4419-9359-5_4,

formula may actually be valid, but the theorem prover was simply not able to prove it. Although such fully automated theorem provers cannot prove theorems whose proofs require real insight, they are very good at quickly grinding through the large and tedious case analysis required in proofs about programs. As a result, fully automated theorem provers are very specialized tools, and as such, they are typically combined with other tools that process the system being verified in various ways to generate the actual queries to the automated theorem prover. For example, SLAM [9] repeatedly sends queries to an automated theorem prover to perform predicate abstraction within a larger outer loop of counter-example guided abstraction refinement. Cobalt [131], Rhodium [133] and PEC [120] attempts to verify the correctness of compiler optimizations by querying a theorem prover with various logical formulas generated from optimizations written in a domain-specific language.

Within fully automated theorem provers, the kinds of formulas that typically arise in program verification are first-order logic formulas that make use of various *theories*, for example the theory of integers, reals, arrays, lists, or bit-vectors. Such formulas are known to be instances of a problem called *Satisfiability Modulo Theories*, or SMT. As a result, so-called *SMT solvers* are the most common kind of fully automated theorem prover used in program verification. Two prominent examples of such SMT solvers include SIMPLIFY [44] and Z3 [152]. For the remaining of this chapter, we first give an overview of SMT solvers (Sect. 4.1), then cover two of the main verification techniques used in combination with SMT solvers, namely Hoare logic (Sect. 4.2) and weakest preconditions (Sect. 4.3), and finally, we cover several additional complexities that arise in realistic programs (Sect. 4.4).

4.1 Satisfiability Modulo Theories

Formulas that typically arise during system verification typically involve several theories. SMT solvers are meant to determine the validity of precisely such formulas. An SMT solver typically has a decision procedure for each theory that it handles. A decision procedure for a theory is simply an algorithm for determining the validity of logical formulas in that theory. Having individual decision procedures, however, is not enough, since the formulas such as the one above span *multiple* theories. An SMT solver must therefore also have an approach for *combining* individual decision procedures. The predominant approach for doing this is the Nelson–Oppen approach [161], used both in SIMPLIFY and Z3. In this approach, the individual theories simply communicate with each other by propagating equalities between variables.

In addition to combining theories, SMT solvers must also have a way to handle quantifiers, namely $\forall$ and $\exists$. The main challenge in doing so is to figure out when and how to instantiate universally quantified assumptions. One of the common techniques for doing this is a heuristic called *matching*. Suppose for example that

$\forall x_1 \ldots x_n.P$ is known to hold. The goal then is to find substitutions θ giving values to $x_1 \ldots x_n$ such that $\theta(P)$ will be useful in the proof. The general idea in matching is to pick a term t from P called a trigger, and to instantiate P with a substitution θ if $\theta(t)$ is a term that the prover is likely to require information about. Each prover has its own way of deciding when it wants information about a term $\theta(t)$. In SIMPLIFY and Z3, the theorem prover checks to see if $\theta(t)$ is represented in a data structure called the E-graph, which stores all the currently know equalities. The intuition of why matching is a good heuristic is that, since P contains the term t, $\theta(P)$ will provide information about $\theta(t)$, and this is likely to be useful since the prover wants information about $\theta(t)$. The choice of a trigger is obviously important. Various heuristic can be used to determine a trigger automatically, for example by picking the smallest term, or set of terms, that cover all the quantified variables. Many SMT solvers also supports user-defined triggers.

We have now seen an overview of SMT solvers. However, a system developer does not typically interact with the SMT solver directly. Instead, an automated tool usually sits between the developer and the SMT solver. Such tools process the system being verified to generate queries to the SMT solver. In doing so, these tools make use of variety of formal techniques to guarantee that the formulas established by the SMT solver really do imply that the system being verified has the properties it is meant to have. We will next cover two of these formal techniques: Hoare Logic and Weakest Preconditions.

4.2 Hoare Logic

Hoare logic (sometimes also called Floyd–Hoare logic) is a formal system introduced by C. A. R. Hoare in 1969 [92], following ideas published earlier by Robert Floyd in 1967 [60]. Hoare logic can be used to reason formally about the correctness of programs by specifying precisely what conditions on entry to a program guarantee what conditions on exit.

Hoare logic makes use of *assertions* to specify conditions at the entry and exit points of a program. An assertion is simply a predicate over a program state. Examples of assertions include: $\mathtt{x} > 5$, $\mathtt{x} = \mathtt{y}$, or $\mathtt{x} = \mathtt{y} + \mathtt{z}$. Each one of these assertions evaluates to either true or false in a program state. For example, in a simple programming language with no heap, the program state simply indicates what value each variables has. Consider the program state where $\mathtt{x}$ has the value 1, $\mathtt{y}$ has the value 2, and $\mathtt{z}$ has the value 3; in this program state, $\mathtt{x} > 5$ does not hold, $\mathtt{x} = \mathtt{y}$ does not hold, and $\mathtt{z} = \mathtt{x} + \mathtt{y}$ holds.

The main judgment of Hoare logic is a *Hoare triple* $\{P\}\ S\ \{Q\}$, where P and Q are assertions as explained above, and S is a program. The Hoare triple $\{P\}\ S\ \{Q\}$ states that if S starts in a state satisfying P, and it terminates, then it terminates in

a state satisfying Q. The assertion P is called the *precondition*, and Q is called the *postcondition*. Some examples of Hoare triples include:

$$\{true\}\ \mathtt{x} := 5\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 0\}\ \mathtt{x} := 5\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 5\}\ \mathtt{z} := 0\ \{\mathtt{x} = 5\}$$
$$\{\mathtt{x} = 1\}\ \mathtt{y} := \mathtt{x} + 1\ \{\mathtt{y} = 2\}$$
$$\{\mathtt{x} \leq 7\}\ \mathtt{x} := \mathtt{x} + 1\ \{\mathtt{x} \leq 8\}$$
$$\{\mathtt{x} = 1 \wedge y \leq 9\}\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2\ \{\mathtt{x} = 2 \wedge y \leq 7\}$$
$$\{\mathtt{x} = \mathtt{y} + \mathtt{z}\}\ \mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2\ \{\mathtt{x} = \mathtt{y} + \mathtt{z} + 3\}$$

One thing to notice about the above examples is that one statement, for example $\mathtt{x} := 5$, can satisfy many Hoare triples. The relation between all the Hoare triples for a given statement is captured by what is known as the *rule of consequence*:

$$\text{if } P' \Rightarrow P \text{ and } \{P\}\ S\ \{Q\} \text{ and } Q \Rightarrow Q' \text{ then } \{P'\}\ S\ \{Q'\} \tag{4.1}$$

This rule states that if we have $\{P\}\ S\ \{Q\}$, then we can always replace P with something that implies P and Q with something that Q implies.

In general, this is just one of many rules that as a whole form Hoare logic. Whereas the above rule of consequence is geared towards relating all the Hoare triples for an *arbitrary* statement S, the other rules in Hoare logic are geared towards defining Hoare triples for the *specific* kinds of statements in a language (for example assignment, conditionals, loops). However, we will actually not go into the details of the other rules of Hoare logic here, because we will instead use a reformulation of Hoare logic based on the notion of *weakest precondition*.

4.3 Weakest Preconditions

Weakest preconditions, which were introduced by Edsger W. Dijkstra in 1975 [45], provide an alternate view of Hoare Logic. Even though Hoare Logic provides rules for showing that a Hoare triple holds, it does not by itself define an algorithm for automatically combining these rules to show that a particular Hoare triple holds. Weakest preconditions, on the other hand, do precisely this: they provide a mechanism for automatically finding a valid way to combine Hoare rules to establish a Haore triple.

However, we're not quite ready yet to define weakest preconditions. Before doing that, we must first define the notion of *stronger* and *weaker* predicates. In particular, if $P \Rightarrow Q$, then we say that P is *stronger* than Q, and Q is *weaker* than P. In essence, P imposes more restrictions on the state, and Q imposes less restrictions. The strongest predicate is *false* (since *false* implies anything) and the weakest predicate is *true* (since any predicate implies *true*).

Having seen what weaker/stronger predicates are, we can now define weakest preconditions. In particular, the *weakest precondition* of a predicate Q with respect to a program S, denoted by $\mathsf{wp}(S, Q)$, is the weakest predicate P such that $\{P\}\ S\ \{Q\}$. More specifically, $\mathsf{wp}(S, Q) = P$ if and only if the following two conditions hold:

$$\{P\}\ S\ \{Q\} \tag{4.2}$$

$$\text{for all } P' \text{ such that } \{P'\}\ S\ \{Q\},\ P' \Rightarrow P \tag{4.3}$$

Immediately from this definition of wp, we see that:

$$\{P'\}\ S\ \{Q\} \text{ if and only if } P' \Rightarrow \mathsf{wp}(S, Q) \tag{4.4}$$

Property (4.4) is at the core of many verification algorithms. Before seeing how this property is used, let's see why it holds. In particular, we'll show the left-to-right direction and then the right-to-left direction of the "if and only if" in (4.4). In both of these directions, let $P = \mathsf{wp}(S, Q)$. In the left-to-right direction, we assume $\{P'\}\ S\ \{Q\}$ and we need to show $P' \Rightarrow P$. From part two of the wp definition, namely condition (4.3), and from $\{P'\}\ S\ \{Q\}$, we immediately get $P' \Rightarrow P$. In the right-to-left direction, we assume $P' \Rightarrow P$ and we need to show $\{P'\}\ S\ \{Q\}$. From part one of the wp definition, namely condition (4.2), we have $\{P\}\ S\ \{Q\}$. Using the rule of consequence (4.1), combined with $P' \Rightarrow P$, we then get $\{P'\}\ S\ \{Q\}$.

To see how property (4.4) can be used for verification, let's assume for now that we have an automated way of computing wp, and that we have an SMT solver of the kind described in Sect. 4.1. Then property (4.4) gives us a way of performing automated program verification: to establish $\{P\}\ S\ \{Q\}$, all we need to do is compute $\mathsf{wp}(S, Q)$, and then ask the SMT solver to show $P \Rightarrow \mathsf{wp}(S, Q)$.

This verification approach sounds simple, elegant, and in the end quite appealing. However, in presenting the approach, we made one huge assumption, which is that wp can be computed. We need to carefully revisit this assumption. After all, the definition of wp given so far is *descriptive*, in that it tells us the properties that wp must satisfy, but it is not *prescriptive*, in that it does not tell us how to compute $\mathsf{wp}(S, Q)$. So, can $\mathsf{wp}(S, Q)$ be computed? Unfortunately, in the general case, the answer is no – computing $\mathsf{wp}(S, Q)$ in general is undecidable. However, it turns out that the main source of undecidability lies in looping constructs, and although loops are important, for the sake of simplicity, we will first present the case without loops, and then discuss loops later.

If we ignore loops, then $\mathsf{wp}(S, Q)$ becomes computable using very simple syntactic rules. Each statement kind in the programming language has an associated rule. We cover here the most important statement kinds: assignment, sequences of statements, and conditionals.

Skip. The simplest weakest precondition rule is for the no-op statement `skip`:

$$\mathsf{wp}(\texttt{skip}, Q) = Q$$

Assignment. Assignment, on the other hand, is more complicated. The weakest precondition for assignments is given by:

$$\mathsf{wp}(X := E, Q) = Q[X \mapsto E]$$

In the above, X is a variable, E is a pure expression with no side-effects, and $Q[X \mapsto E]$ stands for the predicate Q with every occurrence of the variable X replaced with the expression E. To see why this rule works, note that whenever the postcondition Q refers to X, it is referring to the value of X *after* the assignment. Thus, the weakest precondition *before* the assignment is that Q must hold, but on the value of X *after* the assignment. The problem here is that before the assignment, any references to X refer, not surprisingly, to the value of X before the assignment. So to make Q in the precondition refer to the value of X *after* the assignment, we simply replace X with E, its value after the assignment.

Here are several examples of computing the weakest precondition for assignments:

$$\begin{aligned}
\mathsf{wp}(\mathtt{x} := 5, \mathtt{x} = 5) &= 5 = 5 \\
\mathsf{wp}(\mathtt{z} := 0, \mathtt{x} = 5) &= \mathtt{x} = 5 \\
\mathsf{wp}(\mathtt{y} := \mathtt{x} + 1, \mathtt{y} = 2) &= \mathtt{x} + 1 = 2 \\
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1, \mathtt{x} \leq 8) &= \mathtt{x} + 1 \leq 8 \\
\mathsf{wp}(\mathtt{z} := \mathtt{x} + \mathtt{y}, \mathtt{z} \leq 8) &= \mathtt{x} + \mathtt{y} \leq 8
\end{aligned}$$

Sequencing. The weakest precondition for sequences of statements is given by:

$$\mathsf{wp}(S_1; S_2, Q) = \mathsf{wp}(S_1, \mathsf{wp}(S_2, Q))$$

Essentially, we first compute the weakest precondition with respect to S_2, and then with respect to S_1. Here are several examples:

$$\begin{aligned}
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2,\ \ \mathtt{x} = 2 \wedge y \leq 7) &= \mathtt{x} + 1 = 2 \wedge y - 2 \leq 7 \\
&= \mathtt{x} = 1 \wedge y \leq 9 \\
\mathsf{wp}(\mathtt{x} := \mathtt{x} + 1;\ \mathtt{y} := \mathtt{y} - 2,\ \ \mathtt{x} = \mathtt{y} + \mathtt{z} + 3) &= \mathtt{x} + 1 = \mathtt{y} - 2 + \mathtt{z} + 3 \\
&= \mathtt{x} = \mathtt{y} + \mathtt{z}
\end{aligned}$$

Conditionals. The weakest precondition for conditionals is given by:

$$\mathsf{wp}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, Q) = (B \Rightarrow \mathsf{wp}(S_1, Q)) \wedge (\neg B \Rightarrow \mathsf{wp}(S_2, Q))$$

The intuition is that if B holds, then S_1 executes and so $\mathsf{wp}(S_1, Q)$ must hold; if B does not hold, then S_2 executes and $\mathsf{wp}(S_2, Q)$ must hold. For example:

$$
\begin{aligned}
&\mathsf{wp}(\textbf{if } \mathtt{a} < 0 \textbf{ then } \mathtt{a} := -\mathtt{a} \textbf{ else } \mathtt{skip}, \mathtt{a} = 5) \\
&= (\mathtt{a} < 0 \Rightarrow \mathsf{wp}(\mathtt{a} := -\mathtt{a}, \mathtt{a} = 5)) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathsf{wp}(\mathtt{skip}, \mathtt{a} = 5)) \\
&= (\mathtt{a} < 0 \Rightarrow -\mathtt{a} = 5) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathtt{a} = 5) \\
&= (\mathtt{a} < 0 \Rightarrow \mathtt{a} = -5) \wedge (\neg(\mathtt{a} < 0) \Rightarrow \mathtt{a} = 5)
\end{aligned}
$$

This is equivalent to $(\mathtt{a} < 0 \Rightarrow \mathtt{a} = -5) \wedge (\mathtt{a} \geq 0 \Rightarrow \mathtt{a} = 5)$, which in turn, is equivalent to $\mathtt{a} = -5 \vee \mathtt{a} = 5$.

Recap Example. We now show an example that puts together all the statement kinds we've seen so far, and shows how we can use weakest preconditions and an SMT solver to perform program verification.

Say we want to establish the following Hoare triple:

$$
\begin{aligned}
&\{\mathit{true}\} \\
&\quad S_1 : \ \mathtt{a} := \mathtt{x}; \\
&\quad S_2 : \ \textbf{if } \mathtt{a} < 0 \textbf{ then } \mathtt{a} := -\mathtt{a} \textbf{ else } \mathtt{skip}; \\
&\quad S_3 : \ \textbf{if } \mathtt{z} > 0 \textbf{ then } \mathtt{z} := \mathtt{z} - 1 \textbf{ else } \mathtt{skip} \\
&\{\mathtt{a} \geq 0\}
\end{aligned}
$$

The above code computes the absolute value of `x` and stores the result in `a`. It also decrements `z`. The Hoare triple states that after this computation, the value in `a` (which is the absolute value of `x`) must be positive, no matter what the original value of `x` is.

First, we start by systematically computing the weakest precondition with respect to the above program, using the rules we have already seen:

$$
\begin{aligned}
&\mathsf{wp}(S_1; S_2; S_3, \mathtt{a} \geq 0) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2; S_3, \mathtt{a} \geq 0)) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2, \mathsf{wp}(S_3, \mathtt{a} \geq 0))) \\
&= \mathsf{wp}(S_1, \mathsf{wp}(S_2, (\mathtt{z} > 0 \Rightarrow \mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{a} \geq 0))) \\
&= \mathsf{wp}\left(S_1, \begin{bmatrix} (\mathtt{a} < 0 \Rightarrow ((\mathtt{z} > 0 \Rightarrow -\mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow -\mathtt{a} \geq 0))) \wedge \\ (\neg(\mathtt{a} < 0) \Rightarrow ((\mathtt{z} > 0 \Rightarrow \mathtt{a} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{a} \geq 0))) \end{bmatrix}\right) \\
&= \begin{bmatrix} (\mathtt{x} < 0 \Rightarrow ((\mathtt{z} > 0 \Rightarrow -\mathtt{x} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow -\mathtt{x} \geq 0))) \wedge \\ (\neg(\mathtt{x} < 0) \Rightarrow ((\mathtt{z} > 0 \Rightarrow \mathtt{x} \geq 0) \wedge (\neg(\mathtt{z} > 0) \Rightarrow \mathtt{x} \geq 0))) \end{bmatrix}
\end{aligned}
$$

Then, recall that to establish the Hoare triple, our verification strategy is to make use of (4.4). Thus, we ask an SMT solver to show that the precondition implies the weakest precondition, namely:

$$\mathit{true} \Rightarrow \mathsf{wp}(S_1; S_2; S_3, \mathtt{a} \geq 0)$$

The above condition, which is sent to the SMT solver to discharge, is in general called a *verification condition*. That is to say, a verification condition is a condition that, if established, will guarantee the property we are aiming to show about our program. The process of computing the verification condition is typically called *verification condition generation*, sometimes abbreviated as *VCGen*.

In our case, let's take the above verification condition, and convert it into the input format of an SMT solver. For example, in SIMPLIFY's input format, the verification condition would look like:

```
(IMPLIES TRUE
   (AND (IMPLIES (< x 0)
                 (AND (IMPLIES (> z 0) (>= (- 0 x) 0))
                      (IMPLIES (<= z 0) (>= (- 0 x) 0))))
        (IMPLIES (>= x 0)
                 (AND (IMPLIES (> z 0) (>= x 0))
                      (IMPLIES (<= z 0) (>= x 0))))))
```

The above query is actually a rather simple SMT query, and if we send it to any of the prominent SMT solvers, for example Z3 [152] or SIMPLIFY [44], we would get back *valid*, indicating that the condition holds. Thus, using simple syntactic techniques for computing the weakest precondition, combined with an SMT solver, we are able to automatically establish the above Hoare triple. This verification technique of using weakest preconditions combined with an SMT solver is in fact at the core of several important program verification tools such as ESCJava [58] and Boogie [11], and several translation validation tools [116, 117] (which will be covered later in the book).

4.4 Additional Complexities for Realistic Programs

4.4.1 Path-Based Weakest Precondition

In some cases, including many of the techniques described in this book, it is easier to perform weakest preconditions along the paths of the control flow graph (CFG), rather on the syntactic structure of the program. In general, given a point in the CFG, the weakest precondition at that point is the conjunction of all the weakest preconditions along the CFG paths that start at that point. We will make this clearer in just a moment with an example, but before going through an example we must first introduce a new statement kind that is necessary for path-based weakest preconditions.

In particular, to model assumptions along a path (due to conditionals for example), we introduce a special statement **assume** B, where B is a boolean condition. This statement encodes the fact that during verification we are allowed to assume

that B holds at the point where the assume statement is found. An assume statement is useful in verification when B is somehow known to hold from somewhere outside of the verification framework, and we want to allow the verification to take advantage of this. The most common form of assume comes from conditionals: once we are in the true side of a conditional, we can assume that the branch condition holds; similarly, on the false side of a conditional, we can assume that the negation of the branch condition holds. Thus, if we look at a conditional statement **if** B **then** S_1 **else** S_2, and we convert this to a CFG with assume statements, there would be two paths through this statement: (1) **assume** $B; S_1$ and (2) **assume** $\neg B; S_2$.

As with any other statement kind, assume statements also have a weakest precondition rule. In particular:

$$\mathsf{wp}(\textbf{assume } B,\ Q) = B \Rightarrow Q$$

This encodes precisely the idea that we are allowed to assume B when trying to establish the property Q after the statement **assume** B.

Now that we have seen what assume statements are and how to compute their weakest preconditions, let's return to path-based weakest preconditions. We had stated previously that the weakest precondition at a point in the CFG is the conjunction of all the weakest preconditions along the CFG paths that start at that point. As an example, let's try to derive the weakest precondition rule for conditionals that we have already seen using a path-based approach.

So consider the conditional statement **if** B **then** S_1 **else** S_2. As we saw before, there are two paths through this conditional, and so using the path-based approach to weakest preconditions, we would take the conjunction of the weakest precondition along the two paths:

$$\begin{aligned}
&\mathsf{wp}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, Q) \\
&\quad = \mathsf{wp}(\textbf{assume } B; S_1, Q) \wedge \mathsf{wp}(\textbf{assume } \neg B; S_2, Q) \\
&\quad = \mathsf{wp}(\textbf{assume } B, \mathsf{wp}(S_1, Q)) \wedge \mathsf{wp}(\textbf{assume } \neg B, \mathsf{wp}(S_2, Q)) \\
&\quad = B \Rightarrow \mathsf{wp}(S_1, Q) \wedge \neg B \Rightarrow \mathsf{wp}(S_2, Q)
\end{aligned}$$

This coincides precisely with the rule we had previously seen for conditionals. Although we only show one example here, the path-based approach combined with assume statements generalizes to many different kinds of common control-flow, all of which insert assumptions along various paths, for example switch statements, pattern matching, and object oriented dynamic dispatch. In additional, assume statements are useful in many other settings, and so one typically has to have the infrastructure to deal with them anyway. For these reasons, many of the techniques presented in this book in fact use the path-based approach to weakest preconditions.

4.4.2 Pointers

The examples we have seen so far used a very simple language with variables, constants and primitive operations like $+$ and $-$. Realistic languages have additional features such as pointers which complicate the computation of the weakest precondition.

Consider for example $\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=5)$. Here we assume for the moment that pointers such as $\mathtt{y}$ can point only to variables. The regular assignment rules tells us to replace $\mathtt{x}$ with $*\mathtt{y}+1$ in the postcondition, which gives us $*\mathtt{y}+1=5$ (in turn this is equivalent to $*\mathtt{y}=4$). It turns out that this is in fact the correct weakest precondition, and so we may be tempted to think that the regular rules work even in the face of pointers.

However, unfortunately, this is not the case. Note how in the previous example we started with a predicate $x=5$ that did not contain a pointer, but we finished with a predicate $*\mathtt{y}=4$ that does. So let's see what happens when we *start* with a predicate that contains a pointer. For example, consider $\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=*\mathtt{y}+1)$. The regular assignment rule tells us to replace $\mathtt{x}$ with $*\mathtt{y}+1$ in the postcondition, which gives us $*\mathtt{y}+1=*\mathtt{y}+1$, which in turn is equivalent to *true*. In other words, no matter what the starting state is, the postcondition holds. But this is completely wrong: if $\mathtt{y}$ happens to point to $\mathtt{x}$, then the postcondition is $\mathtt{x}=\mathtt{x}+1$, which can *never* be true ($\mathtt{x}$ can never be equal to itself plus one). So at the very least, the precondition should state that $\mathtt{y}$ cannot point to $\mathtt{x}$. In fact, in this example, this is precisely the weakest precondition: $\mathtt{y} \neq \&\mathtt{x}$.

To see why this is the weakest precondition, we can do a case analysis on whether $\mathtt{y}$ points to $\mathtt{x}$ or to some other variable (say $\mathtt{z}$), and then in each case do the weakest precondition using the regular rules. In particular:

$$
\begin{aligned}
&\mathsf{wp}(\mathtt{x} := *\mathtt{y}+1, \mathtt{x}=*\mathtt{y}+1) \\
&= \begin{bmatrix} (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathsf{wp}(\mathtt{x} := \mathtt{x}+1, \mathtt{x}=\mathtt{x}+1)) \wedge \\ (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathsf{wp}(\mathtt{x} := \mathtt{z}+1, \mathtt{x}=\mathtt{z}+1)) \end{bmatrix} \\
&= \begin{bmatrix} (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathtt{x}+1=\mathtt{x}+1+1) \wedge \\ (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathtt{z}+1=\mathtt{z}+1) \end{bmatrix} \\
&= (\mathtt{y}=\&\mathtt{x} \Rightarrow \mathit{false}) \wedge (\mathtt{y}=\&\mathtt{z} \Rightarrow \mathit{true}) \\
&= \mathtt{y} \neq \&\mathtt{x}
\end{aligned}
$$

This example suggests one approach for handling pointers, which is to perform case analysis on all the possible aliasing scenarios. Assuming we are computing $\mathsf{wp}(S,Q)$, let X be the set of variables used in S or Q, and P be the set of variables that are *dereferenced* in S or Q (that is to say the variables p such that $*p$ occurs in S or Q). Note that P is a subset of X. Starting with the original set X, we add to X one fresh variable that is different from all other variables in X, representing the case where a pointer points to none of the variables in the statement or postcondition.

We now have one case for each possible assignment of variables in P to the address of variables in X. Note that there are $|P|^{|X|}$ cases, since there are $|P|$ pointers, each of which can point to $|X|$ variables. For each case, the aliasing becomes clear, and so the weakest precondition is computable as before. The results for all the cases are combined in the same way as in the example above, using implication ($\Rightarrow$) and conjunction ($\wedge$).

Although this approach is feasible, it is complex to implement, it does not generalize easily to other kinds of aliasing such as arrays and dynamically allocated heaps, and it leads to a large number of cases in the weakest precondition formula. For example, if the statement is $*\mathtt{x} := *\mathtt{y} + *\mathtt{z}$, then $|P| = 3$, $|X| = 4$, and so there are $3^4 = 81$ cases.

There is another approach to dealing with aliasing that is much simpler and avoids the above drawbacks. The key insight in this approach is that by making the program data state *explicit*, rather than *implicit*, we can get a simpler formulation of the weakest precondition computations. Indeed, up until now, when we said that a predicate holds (for example $\mathtt{x} = 5$), we left it implicit in what program data state it holds. If the predicate was used as a *postcondition*, then it would hold in the state *after* the statement; if it was used as a *precondition*, then it would hold in the state *before* the statement. We never gave formal names to the program data states before and after the statement, and they never appeared in the formalism. Alternatively, one can make the program data state explicit in the formalism, by remarking that a predicate is simply a unary relation on states, or equivalently a function from states to booleans. We use σ for a program data state, and so a predicate Q is a function that takes a program data state σ and returns the boolean $Q(\sigma)$ stating whether Q holds in that state.

In addition to the change from implicit states to explicit states, we also make use of a function *step*, which implements the forward semantics of statements in the programming language. In particular, given a program data state σ and a program S, $step(S,\sigma)$ returns the program data state that results from executing S starting in state σ.

With this new definition of program data states, and with the new *step* function, we can express the weakest precondition computation as follows:

$$\mathsf{wp}(S,Q)(\sigma) = Q(step(S,\sigma)) \tag{4.5}$$

This basically states that for the weakest precondition $\mathsf{wp}(S,Q)$ to hold on a state σ right before executing S, the postcondition Q must hold on the state resulting from executing S starting in σ – in other words, the weakest precondition holds on precisely those states that make the postcondition hold after executing S.

As an example, we show how to use this new formulation of weakest preconditions to compute $\mathsf{wp}(*\mathtt{x} := \mathtt{y} + 1, \mathtt{y} = 5)$. Because the program data state is now explicit, the predicate $\mathtt{y} = 5$ must be re-expressed as a predicate Q over a state σ. We do this using the *map* theory from SMT solvers. The map theory represents maps from indices to values. It has two operators, *select* and *store*. The term $select(m,i)$ represents the value in map m at index i. The term $store(m,i,v)$ represents a new

map that is identical to m, except that index i is updated to contain v. There are two axioms that govern the theory of maps:

$$i = j \Rightarrow \mathit{select}(\mathit{store}(m,i,v),j) = v$$
$$i \neq j \Rightarrow \mathit{select}(\mathit{store}(m,i,v),j) = \mathit{select}(m,j)$$

The first axiom, a shorter version of which is $\mathit{select}(\mathit{store}(m,i,v),i) = v$, states that if we store a value at a given index, then reading at the same index will return the original value. The second axiom states that if we store a value at a given index, then reading at a *different* index returns the value from the original map, before the store. The map theory is commonly used theory in formal verification, and SMT solvers have become very effective at reasoning about maps.

Returning to our example, we represent a program data state σ as a map from program variables to values, and we use *select*/*store* to model variable reads/writes. For example, the predicate $\mathtt{y} = 5$ would be expressed as $Q(\sigma) = [\mathit{select}(\sigma,\mathtt{y}) = 5]$. Since for simplicity we assume that our programs only have variables and no dynamically allocated memory, a pointer value is simply the name of a variable. For example, a use of $*\mathtt{y}$ would be expressed as $\mathit{select}(\sigma,\mathit{select}(\sigma,\mathtt{y}))$.

We now need to understand more carefully what *step* does. In general, *step* just implements an operational semantics for the language, by simply defining how to run statements in the language. For example, for the assignment $*\mathtt{x} := \mathtt{y}+1$, *step* needs to first read the value of $\mathtt{x}$ – this value, call it v, is the variable in σ where the result of $\mathtt{y}+1$ should be stored. The *step* function should then read $\mathtt{y}$ from σ, add 1 to this value, and then store the result in σ at v. This can be formalized using *select* and *store* as follows:

$$\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma) = \mathit{store}(\sigma,\mathit{select}(\sigma,\mathtt{x}),\mathit{select}(\sigma,\mathtt{y})+1)$$

Here we only show the definition of *step* for the assignment $*\mathtt{x} := \mathtt{y}+1$, but in general *step* is straightforward to define for other cases, even in the presence of pointers, arrays and heaps – *step* just formalizes our common intuition of how statements in the language run.

We are now ready to apply (4.5) to compute the weakest precondition for our example:

$$\begin{aligned}
&\mathsf{wp}(*\mathtt{x} := \mathtt{y}+1, Q)(\sigma) \quad \text{where } Q(\sigma) \;=\; [\mathit{select}(\sigma,\mathtt{y}) = 5] \\
&\quad = Q(\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma)) \\
&\quad = \mathit{select}(\mathit{step}(*\mathtt{x} := \mathtt{y}+1,\sigma),\mathtt{y}) = 5 \\
&\quad = \mathit{select}(\mathit{store}(\sigma,\mathit{select}(\sigma,\mathtt{x}),\mathit{select}(\sigma,\mathtt{y})+1),\mathtt{y}) = 5
\end{aligned}$$

4.4.3 Loops

So far, we have dealt with loop-less code. The Hoare logic rule for loops is:

$$\text{if } \{P \wedge B\}\ S\ \{P\} \text{ then } \{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{\neg B \wedge P\} \tag{4.6}$$

In the above rule, P is a *loop invariant*, which is a predicate that holds at each iteration of a loop. Indeed, $\{P \wedge B\}\ S\ \{P\}$ essentially guarantees that the body S of the loop **while** B **do** S **end** preserves P: if P holds before executing S, it will also hold after executing S, which means it will hold at the beginning of the next iteration. The addition of B in $\{P \wedge B\}\ S\ \{P\}$ accounts for the fact that when we are about to execute S, we just tested the loop condition B and it must have evaluated to true (note that we could alternatively have used an assume statement, as such: $\{P\}$ **assume** $B; S\ \{P\}$).

Rule (4.6) states that if P is a loop invariant, meaning that $\{P \wedge B\}\ S\ \{P\}$ holds, then if P holds going into the loop, P will be preserved throughout the loop and hold after the loop, giving us $\{P\}$ **while** B **do** S **end** $\{\neg B \wedge P\}$. The addition of $\neg B$ is justified by the fact that once the loops exits, we know that B must have just evaluated to false.

The Hoare rule for loops is very simple, but computing weakest preconditions for loops can be challenging. Consider for example:

$$\mathsf{wp}(\textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}, y = 0)$$

According to the definition of wp, we therefore want to compute the weakest P such that:

$$\{P\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{y = 0\}$$

According to rule (4.6), we may be tempted to try $y = 0$ as the loop invariant. Unfortunately $y = 0$ is simply not a loop invariant for this loop. It turns out that the correct loop invariant in this case is $x = y \wedge x \geq 0$, and this is also the weakest precondition P we are seeking. Indeed, note that $x = y \wedge x \geq 0$ is preserved through the loop body:

$$\{x = y \wedge x \geq 0 \wedge x > 0\}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \{x = y \wedge x \geq 0\}$$

As a result, rule (4.6) gives us:

$$\{x = y \wedge x \geq 0\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{\neg(x > 0) \wedge x = y \wedge x \geq 0\}$$

Now note that $\neg(x > 0) \wedge x = y \wedge x \geq 0$ is equivalent to $x \leq 0 \wedge x = y \wedge x \geq 0$, which implies $y = 0$. Thus the rule of consequence (4.1) gives us:

$$\{x = y \wedge x \geq 0\}\ \textbf{while}\ \mathtt{x} > 0\ \textbf{do}\ \mathtt{x} := \mathtt{x} - 1; \mathtt{y} := \mathtt{y} - 1\ \textbf{end}\ \{y = 0\}$$

The important lesson of this example is that to find the weakest precondition, we had to find the right loop invariant, and unfortunately the post-condition $y = 0$ did *not* directly provide us this loop invariant. Instead, we had to *strengthen* the postcondition $y = 0$ to something which, combined with the knowledge that loop exited, would imply $y = 0$. In our case, this strengthening lead us to $x = y \wedge x \geq 0$, but in general, if the language we are dealing with is Turing complete, then it is undecidable to automatically find the weakest loop invariant that is strong enough to establish a given postcondition. For this reason, computing the *weakest* precondition through loops is undecidable.

Now, for verification, we actually don't always need to find the *weakest* precondition. Often, it is enough to just find *some* precondition. Unfortunately, it is even undecidable to just determine whether there exists *some* invariant that is strong enough to establish the postcondition. So in the end, we are left with designing *heuristics* for finding loop invariants, where we use "heuristics" to indicate that these are not algorithms. However, do not be fooled, these techniques often use very principled approaches, founded in well-established theory, including abstract interpretation [39, 40], predicate abstraction [43], and SMT solvers. In fact, there has been a huge amount of work in finding loop invariants, and many verification problems in the end boil down to finding good invariants. Not surprisingly then, the techniques described in Chaps. 7 and 8 will in part cover some simple techniques for finding such loop invariants.