Chapter 3
State Space Exploration

Before we can discuss string analysis and verification techniques for string manipulating programs, we first need to discuss the semantics of string manipulating programs in more detail.

3.1 Semantics of String Manipulation Languages

Semantics of string manipulating programs we defined in Chap. 2 can be formalized as transitions systems. A transition system $T = (I, S, R)$ consists of a set of states S, a set of initial states $I \subseteq S$ and a transition relation $R \subseteq S \times S$. Let us first define the set of states of a string manipulating program written either in the language shown in Fig. 2.1 or the extended language shown in Fig. 2.4. Each program state will correspond to a program location. If we assume that all statements are labeled with unique labels, we can use the labels of the statements to denote the possible program locations. Let us use L to denote the set of program locations (i.e., the set of statement labels). Each string variable will have a value from the set Σ^* and each integer variable will have a value from the set $\mathbb{Z}$. Let us assume that we are given a program with n string and m integer variables. Then, the set of states of the given program is defined as:

$$S = L \times (\Sigma^*)^n \times (\mathbb{Z})^m$$

Let us assume that $l_1 \in L$ denotes the label of the first statement of the program. Then we can define the set of initial states of the program as:

$$I = \{\langle l_1, \epsilon, \ldots, \epsilon, 0, \ldots, 0\rangle\}$$

T. Bultan et al., *String Analysis for Software Verification and Security*,
https://doi.org/10.1007/978-3-319-68670-7_3

where the program counter is initialized to l_1, all string variables are initialized to ϵ (i.e., empty string) and all integer variables are initialized to 0. Since we are assuming all the variables are initialized, and that there is a single initial statement, the set of initial states is a singleton set. It would not be a singleton set if we assume that the variables are not initialized and their initial values are not known.

The transition relation (R) of the program is defined by the semantics of the statements of the program. The semantics of the statements can be inferred from the semantics of the string operations we defined in Chap. 2.

Given a statement labeled l, let $r_l \subseteq S \times S$ denote tuples of program states $(s_1, s_2) \in r_l$ such that, s_1 is a program state at the program location l, and executing statement l in program state s_1 results in the program state s_2. So, r_l denotes the transition relation of the statement l. Then the transition relation of the whole program can be defined as:

$$R = \bigcup_{l \in L} r_l$$

i.e., taking the union of all the transitions defined by each statement of the program gives us the transition relation of the whole program.

Note that, the transition relation of a string manipulating program can be an infinite state system if we do not put a bound on the lengths of the strings. The undecidability of the reachability problem for string programs that we discussed in Chap. 2 is due to unboundedness of the string variables. In our formal model for string manipulating programs, each string variable has an infinite set of possible values (which is the set Σ^*), and hence, set of states of a string manipulating program is infinite.

Since no computer has an infinite amount of memory, bounding all the domains is a practical approach to program analysis and verification. However, when a program is analyzed or verified based on a given bound, obtained results are not guaranteed to hold when the program execution exceeds that bound. So, assuming an infinite state space is a useful assumption if one wants to check a program's behavior for arbitrarily large state spaces.

Let us consider the string program example shown in Fig. 3.1. This program has three string variables `x`, `y` and `z`. The initial state of this program is: $\langle l, x, y, z \rangle = \langle 1, \epsilon, \epsilon, \epsilon \rangle$ which means that initially program counter is 1, and string variables `x`, `y`, `z` are initialized to the empty string ϵ.

The transition relation of the program is defined as the union of the transition relations of its instructions:

$$R = r_1 \cup r_2 \cup r_3$$

```
1: x := "ab";
2: y := "cd";
3: z := x . y;
```

Fig. 3.1 A simple string manipulating program with three string variables

```
1: read x;
2: while (i < length(x)) {
3:       y := y . charat(x, i);
4:       i := i + 1;
5: }
```

Fig. 3.2 A string manipulating program that copies a read string value character by character

and

$$r_1 = \{(\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle)\}$$
$$r_2 = \{(\langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle)\}$$
$$r_3 = \{(\langle 3, ab, cd, \epsilon\rangle, \langle 4, ab, cd, abcd\rangle)\}$$

where we assume that 4 is an implicit halt instruction.

Let us consider the string program example shown in Fig. 3.2. This program reads a string value to the variable x and then it copies the value of string variable x to string variable y one character at a time. The initial state of this program is $\langle l, x, y, i\rangle = \langle 1, \epsilon, \epsilon, 0\rangle$ which means that initially program counter is 1, string variables x and y are initialized to the empty string ϵ, and the integer variable i is initialized to 0.

The transition relation of the program is defined as the union of the transition relations of its instructions:

$$R = r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5$$

We assume that a read instruction can read any possible string value. So, the transition relation for the instruction 1, r_1 consists of an infinite set of transitions, where for each $s \in \Sigma^*$:

$$(\langle 1, \epsilon, \epsilon, 0\rangle, \langle 2, s, \epsilon, 0\rangle) \in r_1$$

Let us consider a state $\langle 2, ab, \epsilon, 0\rangle$ where the value read to variable x is the string "ab". Following transitions are in the transition relation of this string program:

$$(\langle 2, ab, \epsilon, 0\rangle, \langle 3, ab, \epsilon, 0\rangle) \in r_2$$
$$(\langle 3, ab, \epsilon, 0\rangle, \langle 4, ab, a, 0\rangle) \in r_3$$
$$(\langle 4, ab, a, 0\rangle, \langle 2, ab, a, 1\rangle) \in r_4$$
$$(\langle 2, ab, a, 1\rangle, \langle 3, ab, a, 1\rangle) \in r_2$$
$$(\langle 3, ab, a, 1\rangle, \langle 4, ab, ab, 1\rangle) \in r_3$$
$$(\langle 4, ab, ab, 1\rangle, \langle 2, ab, ab, 2\rangle) \in r_4$$
$$(\langle 2, ab, ab, 2\rangle, \langle 5, ab, ab, 2\rangle) \in r_2$$

where we assume that label 5 corresponds to the termination of the loop.

3.2 Explicit State Space Exploration

When the semantics of a program is defined as a transition system (S, I, R), assertion checking corresponds to checking reachability in this transition system.

Let us consider the statements of the program. Each statement l has corresponding transition relation r_l. Using r_l we can also define a $\text{POST} : S \rightarrow S$ function as follows:

$$s_2 = \text{POST}(s_1, l) \Leftrightarrow (s_1, s_2) \in r_l$$

$\text{POST}(s_1, l)$ denotes the state that the program can go by executing statement l at program state s_1. Note that, in the above definition, we are assuming that the transition system is deterministic, i.e., each state has at most one state that can be reached from it after one step of execution. We can generalize to nondeterministic systems if we allow POST function to return a set of states rather than a single state (as we discuss in the next section).

We can also define the POST function for the overall program as follows:

$$s_2 = \text{POST}(s_1) \Leftrightarrow \exists l \in L \ : \ s_2 = \text{POST}(s_1, r_l)$$
$$s_2 = \text{POST}(s_1) \Leftrightarrow (s_1, s_2) \in R$$

We can think of the POST function as computing the post-condition (or post-image) of a given state.

For the string program example shown in Fig. 3.1, we have the following:

$$\begin{aligned}
\text{POST}(\langle 1, \epsilon, \epsilon, \epsilon \rangle, 1) &= \text{POST}(\langle 1, \epsilon, \epsilon, \epsilon \rangle) &&= \langle 2, ab, \epsilon, \epsilon \rangle \\
\text{POST}(\langle 2, ab, \epsilon, \epsilon \rangle, 2) &= \text{POST}(\langle 2, ab, \epsilon, \epsilon \rangle) &&= \langle 3, ab, cd, \epsilon \rangle \\
\text{POST}(\langle 3, ab, cd, \epsilon \rangle, 3) &= \text{POST}(\langle 3, ab, cd, \epsilon \rangle) &&= \langle 4, ab, cd, abcd \rangle
\end{aligned}$$

Similarly, for the string program example shown in Fig. 3.2, we have the following:

$$\begin{aligned}
\text{POST}(\langle 2, ab, \epsilon, 0 \rangle) &= \langle 3, ab, \epsilon, 0 \rangle \\
\text{POST}(\langle 3, ab, \epsilon, 0 \rangle) &= \langle 4, ab, a, 0 \rangle \\
\text{POST}(\langle 4, ab, a, 0 \rangle) &= \langle 2, ab, a, 1 \rangle \\
\text{POST}(\langle 2, ab, a, 1 \rangle) &= \langle 3, ab, a, 1 \rangle \\
\text{POST}(\langle 3, ab, a, 1 \rangle) &= \langle 4, ab, ab, 1 \rangle \\
\text{POST}(\langle 4, ab, ab, 1 \rangle) &= \langle 2, ab, ab, 2 \rangle \\
\text{POST}(\langle 2, ab, ab, 2 \rangle) &= \langle 5, ab, ab, 2 \rangle
\end{aligned}$$

Algorithm 1 REACHABILITYDFS

```
1: Stack := I;
2: RS := I;
3: while Stack ≠ ∅ do
4:     s := POP(Stack);
5:     s' := POST(s);
6:     if s' ∉ RS then
7:         RS := RS∪{s'};
8:         PUSH(Stack, s');
9:     end if
10: end while
11: return RS;
```

3.2.1 Forward Reachability

Let $RS(I)$, or simply RS, denote the set of states that are reachable from the initial states I of a program, i.e.,

$$RS = \{s \mid \exists s_0, s_1, \ldots, s_n \ : \ \forall i < n \ : \ (s_i, s_{i+1}) \in R \ \wedge \ s_0 \in I \ \wedge \ s_n = s\}$$

Using the POST function we can write a simple depth first search algorithm for computing reachable states of a program as shown in Algorithm 1.

For the string program example shown in Fig. 3.1, the set of reachable states RS can be computed using the algorithm shown in Algorithm 1, and the result would be:

$$RS = \{\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle, \langle 4, ab, cd, abcd\rangle)\}$$

For the program shown in Fig. 3.2, the reachable states can be characterized as follows:

$$\begin{aligned}\langle l, s_1, s_2, i\rangle \in RS \ \Leftrightarrow \ & l = 1 \ \wedge \ s_1 = \epsilon \ \wedge \ s_2 = \epsilon \ \wedge \ i = 0 \\ \vee \ & l = 2 \ \wedge \ s_1, s_3 \in \Sigma^* \ \wedge \ s_1 = s_2.s_3 \ \wedge \ i = \texttt{length}(s_2) \\ \vee \ & l = 3 \ \wedge \ s_1, s_3 \in \Sigma^* \ \wedge \ s_1 = s_2.s_3 \ \wedge \ i = \texttt{length}(s_2) \\ \vee \ & l = 4 \ \wedge \ s_1, s_3 \in \Sigma^* \ \wedge \ s_1 = s_2.s_3 \ \wedge \ i = \texttt{length}(s_2) - 1 \\ \vee \ & l = 5 \ \wedge \ s_1 \in \Sigma^* \ \wedge \ s_1 = s_2 \ \wedge \ i = \texttt{length}(s_2)\end{aligned}$$

The set of states S and the set of reachable states RS for the program shown in Fig. 3.2 are infinite. For infinite states spaces, the explicit state exploration approach shown in Algorithm 1 would not terminate, so we need to find a different approach. However, before we address this issue, let us consider backward reachability problem.

3.2.2 Backward Reachability

Similar to the POST function we can also define a PRE function for backward reachability. Even for deterministic systems one state can have multiple states that can reach it in one step, so we need to define the $\text{PRE} : S \rightarrow 2^S$ function as follows:

$$\begin{aligned} s_2 \in \text{PRE}(s_1, l) &\Leftrightarrow (s_2, s_1) \in r_l \\ s_2 \in \text{PRE}(s_1) &\Leftrightarrow \exists l \in L \ : \ s_2 \in \text{PRE}(s_1, r_l) \\ s_2 \in \text{PRE}(s_1) &\Leftrightarrow (s_2, s_1) \in R \end{aligned}$$

We can think of the PRE function as computing the pre-condition (or pre-image) of a state.

For the string program example shown in Fig. 3.1, we have the following:

$$\begin{aligned} \text{PRE}(\langle 2, ab, \epsilon, \epsilon \rangle) &= \{\langle 1, \epsilon, \epsilon, \epsilon \rangle\} \\ \text{PRE}(\langle 3, ab, cd, \epsilon \rangle) &= \{\langle 2, ab, \epsilon, \epsilon \rangle\} \\ \text{PRE}(\langle 4, ab, cd, abcd \rangle) &= \{\langle 3, ab, cd, \epsilon \rangle\} \end{aligned}$$

We can find all states that can reach a particular target state using a depth first search algorithm similar to the one shown in Algorithm 1 that starts from the target state and uses the PRE function to compute backward reachability as shown in Algorithm 2.

Using the Algorithm 2 we can compute the backward reachability set for a given set of states. For the string program example shown in Fig. 3.1, we can compute the following sets:

$$\begin{aligned} BRS(\langle 3, ab, cd, \epsilon \rangle) &= \{\langle 1, \epsilon, \epsilon, \epsilon \rangle, \langle 2, ab, \epsilon, \epsilon \rangle\} \\ BRS(\langle 4, ab, cd, abcd \rangle) &= \{\langle 1, \epsilon, \epsilon, \epsilon \rangle, \langle 2, ab, \epsilon, \epsilon \rangle, \langle 3, ab, cd, \epsilon \rangle\} \end{aligned}$$

Algorithm 2 BACKWARDREACHABILITYDFS(P)

```
1:  Stack := P;
2:  BRS := P;
3:  while Stack ≠ ∅ do
4:      s := POP(Stack);
5:      for s' ∈ PRE(s) do
6:          if s' ∉ BRS then
7:              BRS := BRS∪{s'};
8:              PUSH(Stack, s');
9:          end if
10:     end for
11: end while
12: return BRS;
```

Since assertion verification can be reduced to reachability checks as we discussed earlier, we can use the reachability algorithm above for verifying assertions. This approach is called *explicit state verification* since states of the transition system are visited individually. One of the problems with this approach is, for large state spaces, exploring state space one state at a time is computationally very expensive. In fact, as we observed, for string manipulating programs, the state space is infinite since we allow strings of arbitrary length. For infinite state systems explicit state verification cannot be used to prove absence of errors, but it can be used to prove existence of errors (since a trace that is discovered by explicit state enumeration that leads to an error state proves the existence of an error).

Explicit state verification can be used to guarantee absence of errors in finite state systems. For example, if we bound the variable domains in string programs we can use explicit state verification to explore the whole state space. However, there is another problem. Although depth first search algorithm explores the state space in linear time with respect to the size of the transition system (where the size of the transition system $T = (S, I, R)$ is $|S| + |T|$), the size of the transition system is exponential in the number of variables in the input program. The exponential growth of the state space of programs is called the *state space explosion problem*, and it limits the scalability of explicit state verification techniques for finite state systems.

3.3 Symbolic Exploration

As an alternative to explicit state enumeration we can consider exploring the state space using sets of states. Rather than exploring one state at a time, we will consider exploring sets of states. In order to do this, we need to first generalize the definition of pre and post-condition functions to sets of states as follows: $\textsc{Pre} : 2^S \rightarrow 2^S$, $\textsc{Post} : 2^S \rightarrow 2^S$, where

$$\begin{aligned}
\textsc{Post}(P, l) &= \{s \mid \exists s' \in P \ : \ (s', s) \in r_l\} \\
\textsc{Post}(P) &= \{s \mid \exists s' \in P \ : \ (s', s) \in R\} \\
\textsc{Pre}(P, l) &= \{s \mid \exists s' \in P \ : \ (s, s') \in r_l\} \\
\textsc{Pre}(P) &= \{s \mid \exists s' \in P \ : \ (s, s') \in R\}
\end{aligned}$$

We refer to $\textsc{Post}$ and $\textsc{Pre}$ as post-condition (or post-image) or pre-condition (or pre-image) functions.

For example, for the string program example shown in Fig. 3.1, we have the following:

$$\begin{aligned}
\textsc{Post}(\{\langle 1, \epsilon, \epsilon, \epsilon\rangle\}) &= \{\langle 2, ab, \epsilon, \epsilon\rangle\} \\
\textsc{Post}(\{\langle 2, ab, \epsilon, \epsilon\rangle\}) &= \{\langle 3, ab, cd, \epsilon\rangle\} \\
\textsc{Post}(\{\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle\} &= \{\langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle\}
\end{aligned}$$

The set of states can be infinite and using the set notation we can define the post-condition of an infinite set of states. For example, for the string program example shown in Fig. 3.2, we have the following:

$$\begin{aligned}
\textsc{Post}(\{s \mid s = \langle 3, x, \epsilon, 0\rangle\}) &= \{s' \mid s' = \langle 4, x, \texttt{charat}(x, 0), 0\rangle\} \\
\textsc{Post}(\{s \mid s = \langle 3, x, y, i\rangle\}) &= \{s' \mid s' = \langle 4, x, y.\texttt{charat}(x, i), i\rangle\} \\
\textsc{Post}(\{s \mid s = \langle 4, x, y, \texttt{length}(y) - 1\rangle\}) &= \{s' \mid s' = \langle 4, x, y, \texttt{length}(y)\rangle\}
\end{aligned}$$

3.3.1 Symbolic Reachability

In order to explain forward and backward reachability computations on sets of states, we first define the lattice formed by the sets of states of the transition system.

Symbolic reachability algorithms deal with sets of states rather than individual states. By processing multiple states at the same time, symbolic techniques can converge to an answer with fewer iterations. For example, for the forward reachability analysis, if we want to compute the set of states reachable from the set of initial states, we can first start with the initial states I. Then we can add all the states reachable from initial states and continue adding new states until there is nothing new to add. This is exactly what the depth-first-traversal algorithm shown in Algorithm 1 does, but it does the traversal one state at a time. Symbolic reachability algorithms compute post-condition of a set of states in each iteration instead of computing post-condition of one state at a time. With an appropriate symbolic representation, symbolic algorithms can compute the post-condition of an infinite set of states in a single iteration.

The sets of states of a transition system form a partial order with respect to the set inclusion (i.e., $\subseteq$). The progress in reachability computations can be expressed with respect to this partial order. For the forward reachability computation, we start with I, and if we are using a symbolic representation, in the next iteration we would compute $I \cup \textsc{Post}(I)$. Note that $I \subseteq I \cup \textsc{Post}(I)$. We started with reachable states I and we made some progress by computing a potentially larger set of states in the next iteration. The goal of a forward symbolic reachability algorithm for computing reachable states would be to compute a larger set of states (with respect to the partial order) in each iteration and hopefully converge on the set of reachable states RS after a number of iterations.

These concepts about forward and backward reachability computations can be formalized by defining a lattice formed by the sets of states of the transition system. Given a transition system $T = (S, I, R)$, the power set of S, 2^S forms a complete lattice $(2^S, S, \emptyset, \subseteq, \cup, \cap)$, with the top element $\top = S$, the bottom element $\bot = \emptyset$, intersection $\cap$ as the meet (greatest lower bound) operator, union $\cup$ as the join (least upper bound) operator, and the set containment $\subseteq$ as the partial order. Then, $\textsc{Pre}$ and $\textsc{Post}$ are functions that map elements of this lattice (sets of states) to the elements of this lattice (sets of states).

Let us consider the string program example shown in Fig. 3.1. Here are some of the set of states for this program:

$$
\begin{aligned}
&I && = \{\langle 1, \epsilon, \epsilon, \epsilon\rangle\}\\
&\textsc{Post}(I) && = \{\langle 2, ab, \epsilon, \epsilon\rangle\}\\
&I \cup \textsc{Post}(I) && = \{\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle\}\\
&\textsc{Post}(I \cup \textsc{Post}(I)) && = \{\langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle\}\\
&I \cup \textsc{Post}(I \cup \textsc{Post}(I)) && = \{\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle\}\\
&\textsc{Post}(I \cup \textsc{Post}(I \cup \textsc{Post}(I))) && = \{\langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle\}\\
&I \cup \textsc{Post}(I \cup \textsc{Post}(I \cup \textsc{Post}(I))) && = \{\langle 1, \epsilon, \epsilon, \epsilon\rangle, \langle 2, ab, \epsilon, \epsilon\rangle, \langle 3, ab, cd, \epsilon\rangle\}
\end{aligned}
$$

and here is how these sets are related in terms of the partial order $\subseteq$:

$$I \subseteq I \cup \textsc{Post}(I) \subseteq I \cup \textsc{Post}(I \cup \textsc{Post}(I)) \subseteq I \cup \textsc{Post}(I \cup \textsc{Post}(I \cup \textsc{Post}(I)))$$

Moreover, we can observe the following:

$$
\begin{aligned}
RS &= I \cup \textsc{Post}(I \cup \textsc{Post}(I))\\
RS &= I \cup \textsc{Post}(RS)
\end{aligned}
$$

We see that the set of reachable states RS is the limit of the sequence of states we have been computing using the Post function. RS is greater than or equal to any element in the sequence, and, once we reach RS, the sequence stops increasing with respect to the partial order. We can explain these phenomena using the concept of fixpoints.

3.3.2 Fixpoints

Given a function $\mathcal{F} : 2^S \rightarrow 2^S$, let $\mathcal{F}\ P$ denote the application of function $\mathcal{F}$ to set $P \subseteq S$.

Given a function $\mathcal{F}$, x is called a fixpoint of the function if

$$\mathcal{F}x = x$$

Interestingly, as we show below, reachability properties can be expressed as fixpoints [136].

We use the lambda calculus notation for functions. A function with argument x is written in lambda calculus as follows: $\lambda x\ .\ \mathcal{F}\ x$

Consider the following function:

$$\lambda x\ .\ I \cup \textsc{Post}(x)$$

The set of reachable states RS is a fixpoint of this function, i.e.,

$$RS = I \cup \textsc{Post}(RS)$$

We can see this as follows: First, $RS \supseteq I \cup \textsc{Post}(RS)$ since $I \subseteq RS$, and any state reachable from a reachable state should be reachable itself, i.e., $\textsc{Post}(RS) \subseteq RS$. Next, we need to show that $RS \subseteq I \cup \textsc{Post}(RS)$. According to the definition of RS, the only way a state s can be in RS is, either 1) $s \in I$, or 2) there exists a state in RS that can reach s, which implies that $RS \subseteq I \cup \textsc{Post}(RS)$.

Next, we are going to show that RS is in fact the least fixpoint of this function. I.e., RS is the smallest fixpoint of the function $\lambda x \,.\, I \cup \textsc{Post}(x)$ with respect to the partial order $\subseteq$.

Let $\mu x \,.\, \mathcal{F}\, x$ denote the least fixpoint of $\mathcal{F}$, i.e., the smallest x such that $\mathcal{F}\, x = x$. Then, we claim that:

$$RS = \mu x \,.\, I \cup \textsc{Post}(x)$$

Note that, since RS is a fixpoint of the function $\lambda x \,.\, I \cup \textsc{Post}(x)$, and since $\mu x \,.\, I \cup \textsc{Post}(x)$ is the least fixpoint of the function $\lambda x \,.\, I \cup \textsc{Post}(x)$ we conclude that $\mu x \,.\, I \cup \textsc{Post}(x) \subseteq RS$.

Next, we need to prove that $RS \subseteq \mu x \,.\, I \cup \textsc{Post}(x)$ to complete the proof. Suppose z is a fixpoint of $\lambda x \,.\, I \cup \textsc{Post}(x)$. Then we know that $z = I \cup \textsc{Post}(z)$, which means that $\textsc{Post}(z) \subseteq z$. So, all states that are reachable from z are in z. Since we also have $I \subseteq z$, any path that is reachable from I must also be in z, which means that $RS \subseteq z$.

Since we showed that RS is contained in any fixpoint of the function $\lambda x \,.\, I \cup \textsc{Post}(x)$, it should also be contained in its least fixpoint, since the least fixpoint itself is a fixpoint. So we conclude that $RS \subseteq \mu x \,.\, I \cup \textsc{Post}(x)$ which concludes the proof.

Now, we discuss how to compute the least fixpoint. We call a function $\mathcal{F}$ *monotonic*, if $p \subseteq q$ implies $\mathcal{F}\, p \subseteq \mathcal{F}\, q$. We have the following property from the lattice theory [102]:

Let $\mathcal{F} : 2^S \to 2^S$ be a monotonic function. Then $\mathcal{F}$ always has a least fixpoint, which is defined as

$$\mu x \,.\, \mathcal{F}\, x \equiv \bigcap \{\, x \mid \mathcal{F}\, x \subseteq x \,\}$$

Since $\mu x \,.\, \mathcal{F}\, x$ is the least fixpoint of the function $\mathcal{F}$, it is the intersection (greatest lower bound) of all the fixpoints of $\mathcal{F}$. In fact, it is the intersection of all the fixpoints of $\mathcal{F}$, i.e., it is the intersection of all the sets x where $\mathcal{F}\, x \subseteq x$. This property is valid even when S (hence the lattice) is infinite.

Given a function $\mathcal{F}$, $\mathcal{F}^i\, x$ is defined as:

$$\mathcal{F}^i\, x \text{ is defined as } \underbrace{\mathcal{F}\, (\mathcal{F} \ldots (\mathcal{F}}_{i \text{ times}}\, x)).$$

We define $\mathcal{F}^0$ as the identity relation. Then, we have the following property [102]:

Given a monotonic function $\mathcal{F} : 2^S \rightarrow 2^S$, for all n,

$$\mu x \,.\, \mathcal{F}\, x \supseteq \bigcup_{i=0}^{n} \mathcal{F}^i\, \emptyset$$

This property holds even when the lattice is infinite.

Assume that we generate a sequence of approximations to the least fixpoint $\mu x \,.\, \mathcal{F}\, x$ of a monotonic function $\mathcal{F}$ by generating the following sequence:

$$\emptyset,\ \mathcal{F}\,\emptyset,\ \mathcal{F}^2\,\emptyset,\ \ldots,\ \mathcal{F}^i\,\emptyset,\ \ldots$$

This sequence is monotonically increasing since $\emptyset$ corresponds to the bottom element of the lattice, and the function $\mathcal{F}$ is monotonic. If this sequence converges to a fixpoint, i.e., if we find an i where $\mathcal{F}^i\,\emptyset \equiv \mathcal{F}^{i+1}\,\emptyset$, then from the property above, we know that it is the least fixpoint, i.e., it is equal to $\mu x \,.\, \mathcal{F}\, x$.

Similarly, a monotonically decreasing sequence of approximations could be generated to compute the greatest fixpoint of a function [136]. In this monograph we are focusing on least fixpoints. Because of the duality between the least and the greatest fixpoints, the techniques described here can also be applied to computation of greatest fixpoint.

As an example for computing least fixpoints, consider the computation of the least fixpoint for reachable states: $RS = \mu x \,.\, I \cup \textsc{Post}(x)$. We can compute this least fixpoint by generating the following sequence:

$$\underbrace{\underbrace{\underbrace{\emptyset \vee I}_{\mathcal{F}\,\emptyset} \vee \textsc{Post}(I)}_{\mathcal{F}^2\,\emptyset} \vee \textsc{Post}\,(\textsc{Post}(I))}_{\mathcal{F}^3\,\emptyset} \vee \textsc{Post}\,(\textsc{Post}\,(\textsc{Post}(I))) \vee \ldots$$

When this sequence converges to a fixpoint, the result will be equal to RS. This is exactly the sequence we computed for the string program example shown in Fig. 3.1 above.

In Algorithm 3 we give the fixpoint computation algorithm for the reachable states based on this iterative approach. Note that this fixpoint computation is closely related to the state space exploration algorithm given in Algorithm 1. Both algorithms first add the initial states to the reachable states and then keep adding states that are reachable from the initial states to the reachable states. They both stop when there is no more state left to add (i.e., when exploration reaches a fixpoint). The fixpoint computation algorithm traverses the state space in breadth-first order instead of the depth-first traversal order used in Algorithm 1. Also the fixpoint exploration algorithm processes a set of states at each iteration whereas the explicit state exploration algorithm processes a single state at each iteration.

Algorithm 3 REACHABILITYFIXPOINT

```
1: RS := I;
2: repeat
3:     RS′ := RS;
4:     RS := RS∪POST(RS);
5: until RS = RS′
6: return RS;
```

Algorithm 4 BACKWARDREACHABILITYFIXPOINT(P)

```
1: BRS := P;
2: repeat
3:     BRS′ := BRS;
4:     BRS := BRS∪PRE(BRS);
5: until BRS = BRS′
6: return BRS;
```

We can also compute backward reachability similarly using the PRE function as shown in Algorithm 4.

In order to implement the fixpoint computation algorithms we need a way to represent the sets of states. In general this representation should support tests for equivalence, emptiness, and membership, and meet (intersection) and join (union) operations. If the state space is finite, explicit state enumeration would be one such representation. Note that as the state spaces of the programs grow, explicit state enumeration will become more expensive since the size of this representation is linearly related to the number of states in the set it represents. Unfortunately, as we discussed above, the state spaces of programs increase exponentially with the number of variables. This state space explosion problem makes a naive implementation of the explicit state enumeration infeasible. Moreover, as we have seen for string programs, if we want to represent all possible string values during reachability analysis, then the number of states becomes infinite and an explicit state representation becomes impossible.

The symbolic reachability analysis techniques use a *symbolic representation* for encoding sets of states. Symbolic representations are mathematical objects (such as formulas in some logic) with semantics corresponding to sets of states. We can use such representations in encoding the sets of program states. Using a symbolic representation we can implement the iterative fixpoint computation algorithm and compute the reachable states. As we discuss in the next chapter, in this monograph we mainly focus on use of automata as a symbolic representation for sets of states of string programs.

3.4 Summary

In this chapter we provided a basic survey of reachability analysis for verification of string manipulating programs starting with explicit state enumeration. We discussed both forward and backward reachability analysis using depth-first search where states of a given string manipulating program are traversed one state at a time. Next, we discussed symbolic reachability analysis, where the basic idea is to perform state exploration using sets of states rather than traversing states one by one. We discussed that reachability analysis corresponds to fixpoint computations, and, in order to develop a symbolic analysis framework for string manipulating programs, we need to first develop a symbolic representation that can represent sets of strings. We discuss a symbolic representation for sets of strings in the next chapter.