Lecture Notes: Symbolic Execution

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues
clegoues@cs.cmu.edu

1 Symbolic Execution Overview

In previous lectures, we developed axiomatic semantics as a way to represent a program meaning in terms of what is true after code executes. We used weakest preconditions (versus strongest postconditions) to prove program properties (that is, that some property Q is true after S executes, assuming P is true before it executes). However, because weakest preconditions are typically infeasible to compute, we further extended our language to include loop invariants, and used them to compute *verification conditions* to verify program properties.

1.1 Forward Verification Condition Intuition

Revisiting the verification condition generation procedure, recall that it effectively works *backwards* (which should make sense, given our intuition regarding weakest preconditions). This is visible in the way the recursive calls to $VCGen$ worked. Consider computing the VC for a sequence of assignments with respect to some post condition Q:

$$\begin{array}{l}\{P\}\\ x_1 := e_1\\ x_2 := e_2\\ \{Q\}\end{array}$$

But what if instead, we went *forwards*, after all? We could compute the verification condition for our assignment sequence in a *forwards* manner as follows:

$$\begin{array}{rcl} VC(S,Q) & = & [e_1/x_1]([e_2/x_2]Q) \\ & = & [e_1/x_1, e_2[e_1/x_1]/x_2]Q \end{array}$$

Computing a verification conditions forwards is conducted using a technique known as *symbolic execution.* Symbolic execution is a general technique that executes a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. We introduce it as a mechanism to generate verification conditions; one benefit of symbolic execution is that it makes axiomatic semantics practical. However, it can also be viewed as a way to generalize testing, automate testing, and find bugs in programs (as we will see).

1.2 Formalizing Forward VCGen

We can define forward verification condition generation in terms of symbolically evaluating the program, using the same kinds of big-step rules we used for operational semantics.

Symbolic expressions. We start by defining symbolic analogs for arithmetic expressions and boolean predicates. We will call symbolic predicates *guards* and use the metavariable g, as these will turn into guards for paths the symbolic evaluator explores. These analogs are the same as the ordinary versions, except that in place of variables we use symbolic constants:

$$
\begin{array}{rcl@{\qquad}rcl}
g & ::= & \mathsf{true} & a_s & ::= & \alpha \\
 & | & \mathsf{false} & & | & n \\
 & | & \mathsf{not}\ g & & | & a_{s1}\ op_a\ a_{s2} \\
 & | & g_1\ op_b\ g_2 & & & \\
 & | & a_{s1}\ op_r\ a_{s2} & & &
\end{array}
$$

Next, we generalize the notion of the environment Σ, so that variables refer not just to integers but to symbolic expressions:

$$\Sigma \in \mathit{Var} \to a_s$$

Big-step rules for the symbolic evaluation of expressions results in symbolic expressions. Since we don't have actual values in many cases, the expressions won't evaluate, but variables will be replaced with symbolic constants. That is, we obtain a symbolic expression from a concrete expression e by replacing all variables x in e with their values in the current symbolic state Σ. When these values are concrete, we use the concrete values; if not, we replace variables with symbolic constants:

$$\frac{}{\langle n, \Sigma\rangle \Downarrow n}\ \textit{big-int}$$

$$\frac{}{\langle x, \Sigma\rangle \Downarrow \Sigma(x)}\ \textit{big-var}$$

$$\frac{\langle a_1, \Sigma\rangle \Downarrow a_{s1} \quad \langle a_2, \Sigma\rangle \Downarrow a_{s2}}{\langle a_1 + a_2, \Sigma\rangle \Downarrow a_{s1} + a_{s2}}\ \textit{big-add}$$

Symbolic statement evaluation. We can likewise define rules for statement evaluation. These rules update not only the environment Σ, but also a path guard g associated with that state:

$$\frac{}{\langle g, \Sigma, \texttt{skip}\rangle \Downarrow \langle g, \Sigma\rangle}\ \textit{big-skip}$$

$$\frac{\langle g, \Sigma, s_1\rangle \Downarrow \langle g', \Sigma'\rangle \quad \langle g', \Sigma', s_2\rangle \Downarrow \langle g'', \Sigma''\rangle}{\langle g, \Sigma, s_1; s_2\rangle \Downarrow \langle g'', \Sigma''\rangle}\ \textit{big-seq}$$

$$\frac{\langle a, \Sigma\rangle \Downarrow a_s}{\langle g, \Sigma, x := a\rangle \Downarrow \langle g, \Sigma[x \mapsto a_s]\rangle}\ \textit{big-assign}$$

$$\frac{\langle P, \Sigma\rangle \Downarrow g' \quad g \wedge g' \texttt{SAT} \quad \langle g \wedge g', \Sigma, s_1\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \texttt{if } P \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle} \textit{ big-iftrue}$$

$$\frac{\langle P, \Sigma\rangle \Downarrow g' \quad g \wedge \neg g' \texttt{SAT} \quad \langle g \wedge \neg g', \Sigma, s_2\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \texttt{if } P \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle} \textit{ big-iffalse}$$

The rules for `skip`, sequence, and assignment are compositional in the expected way. The rules for `if` are more interesting. Here, we evaluate the condition to a symbolic predicate g'. In the true case, we use a SMT solver to verify that the guard is satisfiable when conjoined with the existing path condition. If that's the case, we continue by evaluating the true branch symbolically. The false case is analogous.

Doing this produces a symbolic state and a path condition or guard that describes that state. In practice, however, multiple states or paths are possible given symbolic inputs, and so in the general case, symbolic execution in fact produces a *tree* of possible paths/symbolic states guarded by path conditions that lead to them.

Loops. Speaking of multiple paths: what about loops? The simplest way to handle `while` is analogous to `if`, above, but this will lead to a potentially infinite number of paths in the execution tree. Practical tooling (such as what is used in industry, discussed below) typically choose to symbolically "execute" loops up to k times (where k is usually no more than 1 or 2). This approach is particularly applicable if the goal is to find bugs or symbolically enumerate as many paths through the program as possible, especially on real-world code without loop invariants.

When we *do* have loop invariants,[1] we can incorporate them into the verification condition no more than twice: once, the first time the loop invariant is encountered, and again for an "arbitrary" loop iteration. For this arbitrary iteration, we determine which variables could possibly be modified on the path back to the invariant through the loop (how might we do this?). Then, we quantify over a new set of symbolic values for all of those variables (setting them to "arbitrary" values).[2] Symbolic execution can then proceed through the while loop to the rest of the program.

2 Symbolic Execution as a Generalization of Testing

Symbolic execution is fundamentally a way to generalize testing. A test involves executing a program concretely on one specific input, and checking the results. In contrast, symbolic execution considers how the program executes abstractly on a family of related inputs.

2.1 Illustration

Consider the following code example, where a, b, and c are user-provided inputs:

```
1 int x=0, y=0, z=0;
2 if(a) {
3   x = -2;
4 }
```

[1] Note that some approaches or languages for verification support arbitrary invariant annotation to help prune paths or render the verification problem simpler; the approach is the same.

[2] This approach can be similarly applied to extend symbolic execution to function calls.

```
5 if (b < 5) {
6   if (!a && c) { y = 1; }
7   z = 2;
8 }
9 assert(x + y + z != 3);
```

Question: *What is an example input that will lead this assertion to fail? What path is it associated with?*

If we are good (or lucky) testers, we can stumble upon a combination of inputs that triggers the assertion to fail, and then generalize to the combination of input spaces that will lead to it (and hopefully fix it!).

Symbolic execution effectively inverts this process by describing the paths through the program symbolically. Instead of executing the code on concrete inputs (like a = 1, b = 2, and c = 1), it instead tracks execution in terms of symbolic inputs $a = \alpha, b = \beta, c = \gamma$. If a branch condition ever depends on unknown symbolic values, the symbolic execution engine simply chooses one branch to take, recording the condition on the symbolic values that would lead to that branch. We can *split* the state, and use a worklist algorithm to make sure we come back to the other branch.

For example, consider abstractly executing a path through the program above, keeping track of the (potentially symbolic) values of variables, and the conditions that must be true in order for us to take that path. We can write this in tabular form, showing the values of the path condition g and symbolic environment E after each line:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \wedge \beta \geqslant 5 \wedge 0 + 0 + 0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

In the example, we arbitrarily picked the path where the abstract value of a, i.e. α, is false, and the abstract value of b, i.e. β, is not less than 5. We build up a path condition out of these boolean predicates as we hit each branch in the code. The assignment to x, y, and z updates the symbolic state E with expressions for each variable; in this case we know they are all equal to 0. At line 9, we treat the assert statement like a branch. In this case, the branch expression evaluates to $0 + 0 + 0 \neq 3$ which is true, so the assertion is not violated.

Now, we can run symbolic execution again along another path (this style of analysis should feel familiar; note that I haven't given you termination guarantees, however, unlike in abstract interpretation! That's where the slacker vs. non-slacker debate comes back in...). We can do this multiple times, until we explore all paths in the program (*exercise to the reader: how many paths are there in the program above?*) or we run out of time. If we continue doing this, eventually we will explore the following path:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta < 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$
9	$\neg\alpha \wedge \beta < 5 \wedge \neg(0 + 1 + 2 \neq 3)$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$

Along this path, we have $\neg\alpha \wedge \beta < 5$. This means we assign y to 1 and z to 2, meaning that the assertion $0 + 1 + 2 \neq 3$ on line 9 is false. Symbolic execution has found an error in the program!

2.2 Symbolic Execution History and Industrial Use

Symbolic execution was originally proposed (as a way to generalize testing) in the 1970s, but it relied on automated theorem proving, and the algorithms and hardware of that period weren't ready for widespread use. With recent advances in SAT/SMT solving and four decades of Moore's Law applied to hardware, symbolic execution is now practical in many more situations, and is used extensively in program analysis research as well as some emerging industry tools. One of the most prominent examples is the use of the PREfix to find errors in C/C++ code within Microsoft; the Klee symbolic execution engine is well known in open source and research contexts.

Of course, programs with loops have infinite numbers of paths, so exhaustive symbolic execution is not possible. Instead, tools take heuristics, as discussed above. To avoid analyzing complex library code, symbolic executors may use an abstract model of libraries. So, in its most common practical formulations, which uses heuristics scalability and termination, symbolic execution is typically less general than abstract interpretation. However, symbolic execution can often avoid approximating in places where AI must approximate to ensure termination. This means that symbolic execution can avoid giving false warnings; any error found by symbolic execution represents a real, feasible path through the program, and (as we will see) can be witnessed with a test case that illustrates the error.

3 Optional: Heap Manipulating Programs

We can extend the idea of symbolic execution to heap-manipulating programs. Consider the following extensions to the grammar of arithmetic expressions and statements, supporting memory allocation with *malloc* as well as dereferences and stores:

$$\begin{array}{lll} a & ::= & \ldots \mid *a \mid malloc \\ S & ::= & \ldots \mid *a := a \end{array}$$

Now we can define memories as a basic memory μ that can be extended based on stores into the heap. The memory is modeled as an array, which allows SMT solvers to reason about it using the theory of arrays:

$$m \quad ::= \quad \mu \mid m[a_s \mapsto a_s]$$

Finally, we extend symbolic expressions to include heap reads:

$$a_s \quad ::= \quad \ldots \mid m[a_s]$$

Now we can define extended version of the arithmetic expression and statement execution semantics that take (and produce, in the case of statements) a memory:

$$\frac{\alpha \notin \Sigma, m}{\langle malloc, \Sigma, m\rangle \Downarrow \alpha} \textit{ big-malloc}$$

$$\frac{\langle a, \Sigma, m\rangle \Downarrow a_s}{\langle *a, \Sigma, m\rangle \Downarrow m[a_s]} \textit{ big-deref}$$

$$\frac{\langle a, \Sigma, m\rangle \Downarrow a_s \quad \langle a', \Sigma, m\rangle \Downarrow a'_s}{\langle g, \Sigma, m, *a := a'\rangle \Downarrow \langle g, \Sigma, m[a_s \mapsto a'_s]\rangle} \textit{ big-store}$$