Lecture Notes: Program Repair as Reachability

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

Over the past several units, we have moved in this course from addressing *verification* to addressing *synthesis*. The former problem is the task of validating correctness with respect to a specification; the latter is the task of generating a program that meets a required specification. We then moved to applying ideas from synthesis to the problem of program *repair*, or modifying an existing program that fails to meet a provided specification so that it does in fact satisfy it.

There are useful correspondences between techniques we have used for verification and those we then explored for synthesis and repair. For example, verification condition generation began as a method to prove programs correct; generating these conditions forwards rather than backwards allowed us to develop a way to generalize testing through symbolic execution; we could then use symbolic/concolic execution as a way to perform synthesis for repair of certain types of defects.

Today, we will explore a formal connection between program synthesis (for repair, specifically) and verification, formulated as a *reachability* problem.

1 Template-based program synthesis for repair

One general way to formulate program repair as we discussed it last week is as a problem of selecting and appropriately instantiating one or more *repair templates* at the appropriate points in a program. We define a general syntax for a templated program along the following lines (borrowing from the WHILE syntax, but simplifying for the purposes of this discussion):

$$\begin{array}{lcl@{\qquad}lcl}
S & ::= & x := a & a & ::= & a_1 + a_2 \\
 & | & \text{skip} & & | & a_1 - a_2 \\
 & | & S_1;\ S_2 & & | & \boxed{c_i} \quad \textit{called a template parameter!} \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 & & | & \ldots \\
 & | & \text{while } P \text{ do } S & & &
\end{array}$$

Given a templated program with template parameters $c_1 \ldots c_n$ and given template values $\bar{v} = v_1 \ldots v_n$ (corresponding to expressions or constants), we can *instantiate* that emplate on those values to yield a non-templated program. We can define instantiation in a straightforward, syntax-directed way:

$$\begin{array}{lcl}
inst(\text{skip}, \bar{v}) & \rightarrow & \text{skip} \\
inst(S_1; S_2, Q) & \rightarrow & inst(S_1, \bar{v}); inst(S_2, \bar{v}) \\
inst(x := a, \bar{v}) & \rightarrow & x = inst(a, \bar{v}) \\
inst(\boxed{c_i}, \bar{v}) & \rightarrow & v_i
\end{array}$$

[*]These notes draw from Nguyen et al., "Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation", TACAS 2017 [?] and a set of course materials generously provided by Wes Weimer.

The *template-based program synthesis problem* is then defined as follows:

> Given a templated program P with template parameters $c_1 \dots c_n$, and a set T of input-output pairs (tests), do there exist template values $\bar{v} = v_1 \dots v_n$ such that for all $\langle \alpha_0, \beta_0 \rangle \dots \langle \alpha_n, \beta_n \rangle$ in T, $(inst(P, \bar{v}))(\alpha_i) = \beta_i$?

Note that we are ambivalent as to the mechanism used to identify $\bar{v}$, and that many of the inductive techniques we have discussed either for synthesis proper or for program repair specifically fit in this framing (consider syntax-guided synthesis).

We can extend this representation of this problem to program repair by constructing templated programs from the original program and replacing potentially buggy lines with potential template $\boxed{c_i}$. By synthesizing some code to fill arbitrary hole, the repair effectively becomes "delete buggy statement X and replace with instantiated template Y."

For the purposes of this exposition, we focus on single-edit repairs with templates encoding linear combinations of variables; more complicated (e.g., non-linear) templates are usable as well. These templates, like the synthesis-based repair techniques we have discussed, focus on expression-level manipulation.

2 Program Reachability

The problem of reachability as applied to programs asks, very generally, whether given a program P, a set of program variables $x_1 \dots x_n$ and some program label L, do there exist values $c_1 ... c_n$ such that P with $x_i = c_i$ reaches label L in finite time?

We have seen this applied to finding bugs using symbolic execution (e.g., formulating buffer overflows as reaching an error state via program transformation); test generation can be viewed as generating c_i for test inputs with L corresponding to the end of a desired execution path. It is also used in model checking, as we will see in future course lessons.

The following code example revisits the idea/intuition, calling back to our prior discussions on test generation:

```
int x, y; /* global input */
int P() {
if (2 * x == y) {
  if (x > y + 10)
     [ L ]
  return 0;
}
```

Here, $x = -20, y = -40$ reaches the label.

Note that both of these problems are undecidable in general. The "heart" of reachability involves solving all path constraints; each condition makes it harder to find a single consistent set of values. By The "heart" of synthesis is handling all tests, where each test makes it harder to find a single consistent set of values

3 Reducing Synthesis To Reachability

You may recall how reductions work from prior theory or algorithms courses. In brief, Problem A is reducible to Problem B if an efficient algorithm for B could be used as a subroutine to solve

A efficiently. A *gadget* is a subset of a problem instance that simulates the behavior of one of the fundamental units of a different problem.

Thus, given an instance of a synthesis (repair) problem, and assuming we have an oracle that can solve reachability, let us convert the synthesis instance into a reachability instance. If we can do this efficiently, any existing reachability tool/technique (e.g., one that performs symbolic or concolic execution) could be used to repair programs.

Give Q, a template program with a set of template parameters $S = \{c_1, \ldots, c_n\}$ and a set of finite tests $T = (\alpha_1, \beta_1), \ldots$, construct $GadgetS2R(Q, S, T)$ which returns a new program P with a special location L, as follows:

- For every template parameter c_i, add a fresh global variable v_i. A solution to this reachability instance is an assignment of concrete values c_i to variables v_i.
- For every function $q \in Q$, define a similar function $q_P \in P$. The body of q_p is the same as q, but with every reference to a template parameter c_i replace with a reference to the corresponding new variable v_i
- P also contains a starting function $main_P$ that encodes the specification information from the test suite T as a conjunctive expression e:

  $e = \bigwedge_{(\alpha_i, \beta_i) \in T} main_{QP}(\alpha_i) = \beta_i$

  where $main_{qp}$ is a function in P corresponding to the starting function $main_Q$ in Q. In addition, the body of $main_P$ is one conditional statement that leads to a fresh target location L iff e is true.
- P then consists of the declaration of new variables, the functions q_P, and the starting function $main_P$

we turn to slides to demonstrate $GadgetS2R$ for synthesis-to-reachability.

We can also reduce reachability to synthesis, declaring new template variables for each variable; replacing all variables with a read from the template parameter and removing declarations of variables v_i, and raising an exception (or encoding success in some other way) at the location in Q corresponding to the location L in P; by catching the exception and returning some known signal value (like, say, 1), we can encode a single test case for the synthesis problem corresponding to "the function returns 1."

4 Proof of correctness

To prove the correctness of this reduction, we must show that the constructed reachability instance is solvable (with values $c_1 \ldots c_n$) iff the original synthesis instance is solvable (with values $c_1 \ldots c_n$). The reachability instance is solved if those values cause execution to reach L. The synthesis instance is solved if those values cause every test to pass.

The proof uses standard operational semantics to reason about the meaning and executions of programs. As in the past, we use $\Downarrow$ for large-step judgements and $\rightarrow$ for small step. The proof uses large-step semantics to reason about synthesis, which focuses on the final value of the program (its behavior on a test). It uses small-step semantics to reason about reacability, where the intermediate steps matter (was a particular label reached?). It uses induction on the structure of a derivation to show that a property holds fo rall exeuctions of all programs, and weakest preconditions to reason about the special conditional statements that encode test cases.

The high-level proof structure is as follows:

- Lemma 1. The reachability instance method and the synthesis instance method agree on all (non-template) variables.
- Lemma 2. If the reachability instance reaches L from a state S (with values $c_1\ ldotsc_n$), then that state and values model the weakest precondition of the synthesis instance method passing each test.
- Theorem 1. The synthesis instance is solvable iff the reachability instance is solvable (with the same values).

again, slides can demonstrate this correspondence.

4.1 Lemma 1: Agree on Vars

The idea here is to show that the derived program ($p_q(\alpha_i)$) behaves the same as the original program $q[c_q, ..., c_n](\alpha_i)$ when the new variables v_i in P are assigned to the values c_i. Formally:

Let Q be the input synthesis instance method with template variables $v_1 \dots v_n$. Let $P = GadgetS2R(Q)$ be the reachability instance corresponding to method P. For all states E_1, E_2, E_3, all values $c_1, \dots c_n$, all inputs values x, it holds that:

> if $E_1(v_i) = c_i$, then $D_1 :: \langle P(x), E_1\rangle \Downarrow E_2$ iff $D_2 :: \langle inst(Q, \bar{c}), E_1\rangle \Downarrow E_3$ and $\forall y \neq v_i, E_2(y) = E_3(y)$.

The proof proceeds by induction on the structure of the operational semantics derivation D_1. By inversion, the structure of D_1 corresponds exactly to the structure of D_2 except for the template variables.

Case. Suppose D_1 (reachability instance) is:

$$\frac{E_2 = E_1[a \mapsto E_1(v_i)]}{\langle a := v_i, E_1\rangle \Downarrow E_2}\ \textit{assign}$$

By inversion and the construction of P, D_2 is:

$$\frac{E_3 = E_1[a \mapsto c_i]}{\langle a := exp, E_1\rangle \Downarrow E_3}\ \textit{assign}$$

where $exp = inst(\boxed{c_i}, \bar{c}) = c_i$
Now we have $E_2 = E_1[a \rightarrow E_1(v_i)]$ and $E_3 = E_1[a \rightarrow c_i]$
to show: for all $y \neq v_i, E_2(y) = E_3(y)$.
Sub-case 1,$y \neq a$. Then, $E_2(y) = E_3(y)$.
Sub-case 2. $y = a$. To show: $E_1(v_i) = c_i$. This is actually one of the assumptions in the statement of the lemma. (Intuitively, it means the reachability analysis assigned c_i to each variable v_i to reach the label L.)

4.2 Lemma 2 (Reach L = Pass tests)

Let Q be the input synthesis instance method with template variables $v_1 \dots . v_n$ and tests $\langle \alpha_1, \beta_n \rangle$

Let $P =$ GadgetS2R(Q) be the reachability instance method `main`. The execution of P reaches L starting from state E_1 iff $E_1 \models wp(inst(Q, \bar{c})(\alpha_1), result = \beta_1) \wedge \dots wp(inst(Q, \bar{c}(\alpha_n), result = \beta_n)$ where $E_1(v_i) = c_i$

By gadget construction there is only one label L in P, `if` e `then`$[L]$ where e is of the form $f(\alpha_1) = \beta_1 \wedge \dots f(\alpha_n) = \beta_n$.

By standard weakest precondition definitions for if, conjunction, equality and function calls, we have that L is reachable iff $E_1 \models wp(result := f(\alpha_1), result = \beta_1) \wedge \dots wp(result := f(\alpha_n), result = \beta_n)$.

What we *want* is that L is reachable iff $E_1 \models wp(result := inst(Q, \bar{c})(\alpha_1), result = \beta_1) \wedge \dots wp(result := inst(Q, c)(\alpha_n), result = \beta_n)$

So, we have to show that $E_1 \models wp(result := f(\alpha_i), result = \beta_i)$ iff $E_1 \models wp(result := inst(Q, \bar{c})(\alpha_i), result = \beta_i)$. f here is the method from GadgetS2R(Q). By the soundness and completeness of weakest preconditions wrt operational semantics, we have $\langle result := f(\alpha_i), E_1 \rangle \Downarrow E_2$ iff $E_2 \models result = \beta_i$.

By Lemma 1, we have $\langle result := inst(Q, \bar{c})(\alpha_i), E_1 \rangle \Downarrow E_3$ iff $E_1(y) = E_3(y)$ for all $y \neq v_i$.

Since "result" $\neq v_i$, $E_1(result) = E_3(result)$ ("lemma1") and $E_3(result) = \beta_i$ ("Have"), transitively, running the template program Q instantiated with $c_i = v_i$ on a test input produces the required output.

4.3 Correctness

This leads us to the correctness theorem, which is as follows: Let Q be the input synthesis instance method with template variables $v_1 ... v_n$ and tests $\langle \alpha_1, \beta_n \rangle$. Let $P = GadgetS2R(Q)$ be the reachability instance method main.

There exist parameter values c_i such that for all $\langle \alpha_i, \beta_i \rangle$, $inst(Q, \bar{c})(\alpha_i) = \beta_i$ iff there exists input values t_i s.t. the execution of P with $v_i \rightarrow t_i$ reaches L. The proof is from Lemma 2 with $t_i = c_i$.

Note that we can also carry out a constructive reduction going in the other direction. That is, suppose we are given an instance of program reachability. *Can we convert it into a program synthesis instance to solve it?*

5 Implications

Program reachability tools and techniques are much more mature than program repair tools. This correspondence between the problems, among other things, suggests a way to use reachability to attempt to fix bugs in programs. It proceeds by, for every potentially buggy line, in some ranked order, and then for every possible considered repair template, also in some ranked order, converting the repair instance to a reachability instance and then calling an off-the-shelf reachability tool (e.g., an SMT solver-based concolic execution engine like KLEE). If the label is reachable, the discovered parameters in the satisfying model can be returned for instantiation as a program patch.

Overall, this is an instance of an overall correspondence between a variety of the techniques we are considering in this class (e.g., model checking ,test generation, syntax-guided synthesis, and program repair); all of these techniques are seeking, in some way, to statically reason about dynamic execution. We can sometimes take advantage of this correspondence to discover new techniques, as this particular reduction demonstrates.