CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 19: Analyzing Programs

Harvard SEAS - Fall 2021 *Nov. 9, 2021*

1 Announcements

Recommended Reading:

- MacCormick §6.0–6.4
- MacCormick Ch. 3
- Survey article on SMT Solvers: https://cacm.acm.org/magazines/2011/9/122785-satisfiability-modulo-theories/pdf
- Salil's OH this week at normal times, OH next week cancelled since Salil isn't there
- Tuesday: active learning and recorded lecture with live TFs! Thursday: Adam Hesterberg lecturing!
- Participation Portfolio grades almost released, only one for rest of semester

2 SMT Solvers and Program Analysis

Today we will study algorithms for analyzing programs. First, we'll get a quick taste of how a generalization of SAT Solvers, called SMT Solvers (for "Satisfiability Modulo Theories"), are used for finding bugs in programs.

Consider the following program for Binary Search:

```
1  BinarySearch(A, ℓ, u, k)
   Input  : Integers 0 ≤ ℓ ≤ u, a sorted array A of length at least u, a key k
   Output : 1 if k ∈ {A[ℓ], A[ℓ+1], ..., A[u−1]}, 0 otherwise
2  while ℓ < u do
3  |  m = (ℓ + u)/2;
4  |  if A[m] = k then
5  |  |  return 1
6  |  else if A[m] > k then
7  |  |  ℓ = ℓ; u = m
8  |  else ℓ = m + 1; u = u;
9  |  assert 0 ≤ ℓ ≤ u;
10 return 0
```

This looks like a correct implementation of binary search, but to be sure we have added an assert command in Line 9 to make sure that we got all of our arithmetic right and maintain the invariant that $0 \le \ell \le u$. Now to test for bugs, we could run the program on many different inputs

and see if the assert command ever fails. But there are infinitely many choices for the inputs, and even for the pair (ℓ, u) of bounds there are roughly 2^{64} choices if they are 32-bit words, which is infeasible to exhaustively enumerate.

SMT Solvers allow us to more efficiently search for inputs that would violate a condition (or reach a certain state) in a program. To encode our program as an SMT formula, we will have two kinds of variables — propositional variables, which take on boolean values just like in SAT, and "theory variables," which in this case take on integer values like the variables in our program.

For our `BinarySearch()` program here, we will consider whether there are inputs that make the assertion fail *within the first two iterations of the loop.* (The approach can be generalized to any finite number of loop iterations, with a corresponding blow-up in the size of the SMT instance.) To do this, we will have the following variables in our SMT formula:

- x_i for $i = 2, \ldots, 10$: propositional variable representing whether or not we execute line i in the first iteration of the loop.
- x'_i for $i = 2, \ldots, 10$: propositional variable representing whether or not we execute line i in the second iteration of the loop.
- x_f: propositional variable representing whether the assertion fails during the first or second iteration of the loop. (So x_f will be false if the program either halts with an output during the first two iterations or reaches the end of the second iteration without the assertion failing.)
- ℓ, u, k: integer variables representing the input values for ℓ, u, k
- ℓ', u': integer variables representing the values of ℓ, u if and when Line 9 is reached in the first iteration of the loop.
- ℓ'', u'': integer variables representing the values of ℓ, u if and when Line 9 is reached in the second iteration of the loop.
- m, m'; integer variables representing the values assigned to m in the first and second iterations of the loop.
- a, a': representing values of $A[m]$ and $A[m']$.

We then construct our formula as the conjunction of the following constraints, corresponding to the input preconditions (Constraint 1–2), the control flow and assignments made by the program (Constraints 3–29)), and asking for the assertion to fail (Constraint 30).

1. $(0 \le \ell) \wedge (\ell \le u)$
2. $((m \le m') \to (a \le a'))$ $\wedge\, ((m \ge m') \to (a \ge a'))$
3. (x_2)
4. $(x_2 \wedge (\ell < u)) \to x_3$
5. $(x_2 \wedge \neg(\ell < u)) \to x_{10}$
6. $x_3 \to ((m = (\ell + u)/2) \wedge x_4)$
7. $(x_4 \wedge (a = k)) \to x_5$
8. $(x_4 \wedge \neg(a = k)) \to x_6$
9. $x_5 \to \neg x_f$
10. $(x_6 \wedge (a > k)) \to x_7$
11. $(x_6 \wedge \neg(a > k)) \to x_8$
12. $x_7 \to ((u' = m) \wedge (\ell' = \ell) \wedge x_9)$
13. $x_8 \to ((\ell' = m + 1) \wedge (u' = u) \wedge x_9)$
14. $(x_9 \wedge \neg((0 \le \ell') \wedge (\ell' \le u')) \to x_f$
15. $(x_9 \wedge (0 \le \ell') \wedge (\ell' \le u')) \to x'_2$
16. $x_{10} \to \neg x_f$
17. $(x'_2 \wedge (\ell' < u')) \to x'_3$
18. $(x'_2 \wedge \neg(\ell' < u')) \to x'_{10}$
19. $x'_3 \to ((m' = (\ell' + u')/2) \wedge x'_4)$
20. $(x'_4 \wedge (a' = k)) \to x'_5$
21. $(x'_4 \wedge \neg(a' = k)) \to x'_6$
22. $x'_5 \to \neg x_f$
23. $(x'_6 \wedge (a' > k)) \to x'_7$
24. $(x'_6 \wedge \neg(a' > k)) \to x'_8$
25. $x'_7 \to ((u'' = m') \wedge (\ell'' = \ell') \wedge x'_9)$
26. $x'_8 \to ((\ell'' = m' + 1) \wedge (u'' = u') \wedge x'_9)$
27. $(x'_9 \wedge \neg((0 \le \ell'') \wedge (\ell'' \le u'')) \to x_f$
28. $(x'_9 \wedge (0 \le \ell'') \wedge (\ell'' \le u'')) \to \neg x_f$
29. $x'_{10} \to \neg x_f$
30. x_f

Each of these constraints can be turned into a small CNF formula whose variables are either propositional variables or propositions asserting (in)equalities involving the theory variables, like $(m = (\ell + u)/2)$ or $\neg(\ell' \leq u'')$. (Recall that every boolean function on k variables can be written as a k-CNF. Each of the constraints above involves at most 4 propositions.) Taking the AND of all of these CNFs yields a larger CNF that is our SMT instance φ.

A satisfying assignment to the SMT instance will provide an assignment to the propositional variables and the theory variables that makes all of the constraints true, and in particular makes the propositional variable x_f true, signifying that the assertion failed.

To apply an SMT Solver, however, we need to select a "theory" that tells us the domain that the theory variables range over and how to interpret the operations and (in)equality symbols. If we use the standard theory of the natural numbers above, then we will find out that φ is *unsatisfiable*, because Algorithm 10 is a correct instantiation of Binary Search over the natural numbers.

However, if we implement Algorithm 10 in C using the `unsigned int` type, then we should not use the theory of natural numbers, but use the *theory of bitvectors*, because C `unsigned int`'s are 32-bit words, taking values in the range $\{0, 1, 2, \ldots, 2^{32} - 1\}$, with modular arithmetic. And in this case, an SMT Solver will find that the formula is *satisfiable*! One satisfying assignment will have:

1. $\ell = 2^{31}$
2. $u = 2^{31} + 2$
3. $m = (\ell+u)/2 = ((2^{31}+2^{31}+2) \bmod 2^{32})/2 = 1$
4. $a = 0$, $k = 1$
5. $\ell' = \ell = 2^{31}$
6. $u' = m + 1 = 2$

This violates the assertion that $\ell' \leq u'$ — a genuine bug in our implementation of binary search!

3 Program Analysis

Can we use SMT Solvers to do all of our debugging for us? A dream (especially for TFs in CS courses!) is that we could just write a mathematical specification of what our program P should do, such as the following for binary search:

$$\text{Spec} : \forall A, \ell, u, k \;\; (0 \leq \ell \leq u, \forall i \;\; A[i-1] \leq A[i]) \rightarrow (P(A, \ell, u, k) = 1 \leftrightarrow \exists i \in [\ell, u] \;\; A[i] = k)\,.$$

Dream: an algorithm V that given a specification Spec and a program P, $V(\text{Spec}, P)$ will always tell us whether or not P satisfies Spec.

Unfortunately, there are no such general-purpose verification tools. All verification tools like SMT Solvers have one or more of the following limitations:

1. They apply to only a limited class P of programs (e.g. ones annotated by programmers with appropriate assertions, ones without unbounded loops, ones that only use certain data types and operations)
2. They apply only to a limited class of specifications (e.g. checking that specific assertions fail as above, type-safety, etc.)
3. They can find some bugs, but may fail to find bugs even on incorrect programs.
4. They can sometimes find a proof of satisfying a specification, but may fail to find proofs even for correct programs.

One of our goals over the next few lectures is to understand why we haven't achieved this dream of program verification.

4 Universal Programs

What kinds of computations *can* we do on *general* programs? One thing we can do is simulate them.

Theorem 4.1.

1. *There is a RAM program U such that for every RAM program P and input x, we have $U(P, x) = P(x)$. Moreover,* $\text{Time}_U((P, x)) = O(\text{Time}_P(x))$.
2. *There is a Word-RAM program U such that for every Word-RAM program P and input x, we have $U(P, x) = P(x)$. Moreover,* $\text{Time}_U((P, x)) = O(\text{Time}_P(x))$.

Proof idea.
Problem Set 4 RAM simulator! Plus compiling Python down to RAM or Word-RAM

□

We can also write a Word-RAM program that simulates RAM programs and vice-versa.

Q: What would change in the theorem if we want U to be a Word-RAM program but allow P to be a RAM program?

We could have U convert P to a Word-RAM program and simulate it using the Word-RAM simulator above, or convert the RAM simulator U to a Word-RAM program. Either way, though, the runtime below-up in the simulation can be much larger because a RAM program P can construct numbers whose bitlength is exponential in its runtime (as seen in PS4) and simulating computations on these numbers using finite-sized words will take exponential time. So we'd only be able to show something like:

$$\text{Time}_U((P, x)) = 2^{O(\text{Time}_P(x))}.$$

Importance of Universal Programs:

- Historically: Universal Turing Machine (Turing, 1936)
- Hardware vs. Software: Can build just one computer (U) and use it to execute any program P we want. Previously: build new hardware for every new type of problem we want to solve.
- Inspired the development of modern computers (e.g. the "von Neumann Architecture").
- Programs vs. Data: we can think of programs P as data themselves.

Why are we using RAM/Word-RAM?

- Simple, mathematically precise model of computation. Allows all of our results in the course to be actual theorems.
- Clarify what counts as a "single step" when analyzing runtime.
- Corresponds well to how our computers are actually built.
- Captures all reasonable models of computation (Church–Turing Thesis, next week).

5 The Halting Problem

Q: Problem with the Universal Program for program analysis? If P doesn't halt, then U doesn't halt. If we've been simulating $P(x)$ for a long time, we don't know whether $P(x)$ will eventually produce an output or or not.

This motivates trying to solve the following problem:

Input : A RAM program P and an input x
Output : `accept` if P halts on input x, `reject` otherwise

Computational Problem Halting Problem

Theorem 5.1. *There is no algorithm (modelled as a RAM program) that solves the Halting Problem.*

Proof. Suppose for contradiction that there is a RAM program H that solves the halting problem.

First we claim we can use H, together with the universal RAM program U, to construct a RAM program R that solves the following weird problem:

Input : A RAM program P
Output : `accept` if $P(P) =$ `reject`, `reject` otherwise

Computational Problem RejectSelf

Below is pseudocode for a RAM program R that solves RejectSelf:

```
1 R(P):
  Input  : A RAM program P
  Output : accept if P(P) = reject, reject otherwise
2 if H(P, P) = accept then
3 |   if U(P, P) = reject then return accept;
4 |   else return reject;
5 else return reject;
```

Importantly, H will always halt on (P, P) because it is required to return `accept` or `reject`, and if H returns `accept`, then we know that P halts on P so $U(P, P)$ will halt and return $P(P)$.

Now let's consider what happens if we feed R to itself. Since R solves RejectSelf, we have:

$$R(R) = \texttt{accept} \quad \Leftrightarrow \quad R(R) = \texttt{reject},$$

which is clearly a contradiction. So our assumption that there exists a RAM program H solving the holting problem must have been incorrect. □

The self-contradictory nature of R is very related to the paradoxical sentence "This sentence is false" and also the paradoxical set of all sets that don't contain themselves $\{S : S \notin S\}$. It comes out of a "diagonalization" argument very similar as the one used to prove that the real numbers are uncountable.