CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Lecture 23: Uncomputability and Program Analysis

Harvard SEAS - Fall 2022 *2022-11-22*

1 Announcements

Recommended Reading:

- MacCormick Ch. 3 and §7.7–7.9
- Survey article on SMT Solvers: https://cacm.acm.org/magazines/2011/9/122785-satisfiability-modulo-theories/pdf

Announcements:

- Last Sender–Receiver Exercise next Tue 11/29
- Happy Thanksgiving!
- Adam's OH today: Zoom only, https://harvard.zoom.us/my/ahesterberg

2 The Halting Problem

Last time we stated our first example of an *unsolvable problem*, for which there is no algorithm whatsoever:

Input	**:** A RAM program P and an input x
Output	**:** `yes` if P halts on input x, `no` otherwise

Computational Problem Halting Problem

Today we'll prove that it is in fact unsolvable:

Theorem 2.1. *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:

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

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

□

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.

This proof of unsolvability of the Halting Problem is another example of a proof of unsolvability by reduction. That proof can be broken into two parts:

1.

2.

With Lemma 4.2 from last lecture, these two claims imply that the Halting Problem is unsolvable.

3 Other Unsolvable Problems

The phenomenon of unsolvability is not limited to problems about programs. Indeed, there's a sense in which *almost all* computational problems are unsolvable: it can be shown that three are "uncountably many" decision problems over any infinite set $\mathcal{I}$ of inputs, but there are only "countably many" RAM programs P, so most decision problems cannot be solved by any RAM program.

But it's more interesting and useful to identify natural examples of unsolvable problems. One striking example is the following:

Input	**:**
Output	**:**

Computational Problem Diophantine Equations

Example input:

Algorithm for univariate polynomials $p(x) = ax^2 + bx + c$ of degree 2:

Hilbert's 10th Problem: find an algorithm to decide solvability of Diophantine Equations in general. This problem was posed by Hilbert in a famous address to the International Congress of Mathematicians in 1900, as part of a list of 23 problems that Hilbert laid out as challenges for mathematicians to tackle in the 20th century. Several of Hilbert's problems were part of a general project to fully formalize and mechanize mathematics. Hilbert's 2nd problem was to prove consistency of the axioms of mathematics, which was shown to be impossible by Gődel's Incompleteness Theorem in 1931. Turing's work on the undecidability of the Halting Problem was also motivated by Hilbert's program, and was used by Turing to show that there is no algorithm to decide to

truth of well-defined mathematical statements (since whether or not a Turing machine halts on a given input is a well-defined mathematical statement), resolving Hilbert's Entscheidungsproblem ("Decision Problem," posed by Hilbert and Ackermann in 1928). Hilbert's 10th problem asks about a very special case of the Entscheidungsproblem, and was finally resolved in 1970 through the work of several mathematicians:

Theorem 3.1 (Matiyasevich, Robinson, Davis, Putnam)**.** *Diophantine Equations is unsolvable.*

Another very simple unsolvable problem is the following:

Input	**:**
Output	**:**

Computational Problem Tiling

Theorem 3.2. *Tiling is unsolvable.*

4 Program Verification and Analysis

A dream (especially for TFs in CS courses!) is that we could just write a mathematical specification of what a program P should do, such as the following, and then automatically verify that a program meets the specification:

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

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.
Q: Why is the dream not achievable?

Thus all program verification tools have one or more of the following limitations:

Nevertheless, the tools for program verification have grown in power and sophistication over the years, and we will study one of the most powerful approaches.

5 SMT Solvers and Program Analysis

We will see 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 : yes if k ∈ {A[ℓ], A[ℓ + 1], ..., A[u − 1]}, no otherwise
2  while ℓ < u do
3  |  m = (ℓ + u)/2;
4  |  if A[m] = k then
5  |  |  return yes
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 no
```

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 $0 \leq \ell \leq u$. 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 $\neg(\ell' \le u'')$. (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 $\ell' \le u'$ — a genuine bug in our implementation of binary search!

Next time we will describe Satisfiability Modulo Theories more generally and formally and also sketch how the Cook–Levin Theorem can be proven using Satisfiability Modulo the Theory of Bitvectors as an intermediate step.