Undecidability of Program Properties

Aravind Machiry
amachiry@purdue.edu

1 Credits

Copied from Halley Young (UPenn)

2 Program Analysis

Program analysis concerns algorithms for checking properties of programs. In this article, we elucidate the fundamental limits of program analysis by using the language of the theory of computation.

We begin by showing how program analysis can be cast as a decision problem with a Yes/No answer. We next provide intuition for why this decision problem is likely to be a hard problem. We then formalize this intuition by proving a prototypical program analysis problem, called the Halting Problem [1], undecidable. We then provide a recipe for proving other specific program analysis problems – such as whether two programs are semantically equivalent – undecidable by reduction to the Halting Problem. Lastly, we introduce and prove Rice's theorem, a general and powerful result that formalizes the notion that checking any non-trivial property of programs is undecidable.

While this article establishes a fundamental negative result about program analysis, however, it is also what justifies the rich diversity of program analysis techniques in theory and practice, with different tradeoffs.

2.1 Mapping to a decision problem

Analyzing programs is a task that software engineers must perform every day. Whether trying to confirm that their own program works as intended or trying to understand the intentions behind the code of another programmer, software engineers are expected to be able to look at a piece of executable code and state what its intended behavior is.

We can formalize the notion of a program's intended behavior using the language of the theory of computation, where we generalize as much as possible while remaining as precise as possible. Thus, while we could pose program analysis as a problem about programs of a particular form and wherein we want to know a specific property, we instead say that program analysis can be done with any program P (whether or not it has a for loop, terminates, uses recursion, etc.), and for any property ϕ (whether it describes aspects of the run-time heap, termination of the program, or the existence in the program of a divide-by-zero operation). Our general problem formalization is as follows:

Given a program P and a property ϕ, a program analysis must decide in finite time whether or not P satisfies ϕ, returning True if P satisfies ϕ and False if P doesn't satisfy ϕ.

2.2 Example Program Properties

There are many properties that could be considered important, and some are only relevant for certain programs, such as the race freedom property for concurrent programs. However, there are two classes of properties which are always interesting: safety and liveness [2].

Safety: A safety decider returns True if a program will never crash, and False otherwise. It is obvious that such a decider would be useful, but it is less obvious just how powerful it would be. Consider the program below:

```
void main()
{
  unsigned int a, b, c;
  scanf("%d %d %d", a, b, c);

  if (a + b + c != 0 && pow(a,3) + pow(b,3) == pow(c,3))
  {
    assert(false);  // crash
  }
}
```

It is clear that this program is a simple, valid C program. So if we had a safety decider, it would need to return True if the program crashed, and False otherwise.

On closer inspection, however, we see that there is something special about this program. The if statement guards against an instance in which Fermat's Last Theorem [3] is violated. Thus, if our safety decider returned True, we would know that at least a special case of Fermat's last theorem was true. and if it returned False, we would know that Fermat's Last Theorem could not be true. In fact, many mathematical theorems can be written as assertions which would cause a program to crash if true. At this point, it may appear that we can get rid of traditional mathematical theorems and just throw every problem we've got at a safety decider! This is especially powerful in cases such as Fermat's last theorem, which was only proved after much work by the greatest minds in mathematics.

Termination: This is another interesting property, where a decider would determine whether any program would terminate or not. This would certainly be useful for software engineers, who would no longer have to worry about accidentally shipping code with an infinite loop. It would also be a great source of power for mathematics. Just as unproved math theorems can be posed as programs which may or may not violate an assertion, other unproved math theorems can be posed as programs which may or may not run forever.

Termination is just one example of a set of desirable properties called liveness properties. Informally, liveness means that "at some point in running this program, something good will happen." The most obvious thing we'd want to happen with a program is to have it return a value (terminate), but in other instances, such as with programs which are designed to run indefinitely, we may define making progress in some other way. e.g., No deadlocks. For example, consider the following program:

```
int collatz(int n)  // assume n is positive integer
{
  if (n == 1) return 1;
  if (n % 2 == 0) return collatz(n/2);
    return collatz(3*n + 1);
}
```

Does the program terminate for all positive integers? Surprisingly, no one knows – while all numbers tested so far lead to termination, nobody has formally proved that there does not exist some number which causes the program to run infinitely. This program implements the famous Collatz conjecture [4]. If we had a termination decider, we could input this program into it, and find the answer to this conjecture.

The fact that program analysis, if it is always decidable, would virtually eliminate the need for traditional mathematics may sound fantastic. However, in fact it should be viewed as a hint that program analysis should not be decidable in its most general sense. Basically, the (informal) argument goes like this: we can't have the kind of program analysis we'd like as working software engineers without acquiring mathematical superpowers as well, and we're pretty sure no such superpowers exist, so therefore it is impossible to obtain the program analysis tools we'd most desire, even for non-theoretically driven programming.

In the following section, we will abandon this informal argument in favor of a rigorous understanding about the undecidability of these program properties.

2.3 Undecidability

The origins of the first formal proof for the lack of a decision procedure, or undecidability, of certain problems was first discovered in the 1930's, and it was in the context of trying to understand mechanical mathematical reasoning. In 1936, Alan Turing introduced a model of computation known as the Turing Machine [5], which is a theoretical construction that, according to the Church-Turing thesis [6], is at least as powerful in what it can or cannot compute as any modern computer or programming language, or any other imaginable model. That same year, he published about a very simple property of a program-input pair to a Turing machine which is not decidable: whether that program will ever halt, or terminate, given that input.

We will make only one assumption: that a program H exists which, given two inputs – the code of any program P denoted $CODE(P)$ and program input x – will output True if $P(x)$ halts, and False otherwise. From this one assumption we will derive a contradiction, suggesting that no such program can ever exist.

Consider the following program:

```
void D(x)
{
  if (H(x,x) == true) {
    while (true) ; // if it gets here, it never halts!
  } else
    return;        // if H(x,x) is false, it halts
}
```

Now consider feeding the program code D together with another copy of the code of D as input into H. Does $D(CODE(D))$ halt? Let's find out.

Using our definition of H from before and substituting in this hypothetical input, we get that $H(CODE(D), CODE(D))$ is true iff $D(CODE(D))$ halts.

Suppose that $D(CODE(D))$ doesn't halt.

Then, according to our definition of H, $H(CODE(D), CODE(D))$ is false, which according to our definition of D means $D(CODE(D))$ halts.

On the other hand, suppose $D(CODE(D))$ halts.

Then, according to our definition of H, $H(CODE(D), CODE(D))$ is true, which according to our definition of D means $D(CODE(D))$ never halts.

Either way, we arrive at a contradiction and, since the only assumption we made was that H exists, we are forced to concede that a program which decides halting for any given program on any given input cannot exist.

2.4 Reduction to Halting Problem

The halting problem is the most famous example of an undecidable problem. Having this one specific example is exceptionally useful, because through a process called reduction we can rely on the fact that the halting problem is undecidable to show that other useful program analyses are undecidable as well.

Consider the following problem, known as the program equivalence problem: Given a program P and a program W which each take one input, return True if P and W return the same value whenever given the same input, and False otherwise.

Such a decider would be useful in practice, for instance, in checking whether an optimized version of a program is indeed identical to the original unoptimized version. We will show through reduction to the halting problem that this problem is undecidable.

Assume what we really want to know is whether program M halts on input x. Consider the following two programs:

```
int main1(int a} {
  return 1;
}

int main2(int a} {
  M(x);
  return 1;
}
```

Since we know that main1 will always return 1 for all input, and `main2` will always return 1 for all input if and only if M halts on x, by asking "Do `main1` and `main2` output the same value for all inputs?," we obtain the answer to "does M halt on x"? Notice that we can do this for any M and any x, so it is easy to turn a program equivalence decider PE into a halting decider H. Since H doesn't exist, PE can't either.

Let's consider another problem we talked about earlier: program termination. To remind us of the problem: Given a program P which takes some number of input variables (for simplicity we will say one), return True if P terminates on all inputs, and False otherwise.

We will show through reduction to the halting problem that this program is undecidable. Assume what we really want to know is whether program M halts on input x. Construct the program `main2` above. By asking "does `main2` halt on all inputs?," we are really asking if M halts on x. So the existence of a program which could answer this question would effectively give us a program which could solve the halting problem, which we know is impossible. Thus, a termination decider cannot exist.

As a final example, consider a decider for array index out-of-bounds errors: given a program P, return True if P does not index any array out-of-bounds on any input, and False otherwise. Once again, we will show through reduction to the halting problem that this program is undecidable. Assume what we really want to know is whether program M halts on input x. Construct the following program:

```
void main()
{
  int a[1]; // create an array of size 1
  M(x);
  a[2]; // array index out-of-bounds
}
```

By asking "does main index any array out-of-bounds on any input?", we are really asking if M halts on x. So the existence of a program which could answer this question would effectively give us a program which could solve the halting problem, which we know is impossible. Thus, an array index out-of-bounds decider cannot exist.

References

[1] Leslie Burkholder. The halting problem. *ACM SIGACT News*, 18(3):48–60, 1987. Publisher: ACM New York, NY, USA.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, September 1987.

[3] Fermat's Last Theorem, June 2021. Page Version ID: 1028363378.

[4] Collatz conjecture, June 2021. Page Version ID: 1029441799.

[5] Turing machine, June 2021. Page Version ID: 1026352516.

[6] Church–Turing thesis, June 2021. Page Version ID: 1027169790.