Rice's Theorem

This week, we have studied a multitude of undecidable languages, most of them involving the analysis of different models of computation. After all of this, it may seem that most decision procedures that analyze the behavior of machines, in particular, Turing machines may be undecidable. Rice's Theorem answers this in the affirmative:

Theorem 1. *(Rice's Theorem) Let P a property about Turing machines, formalized as a language consisting of Turing machine descriptions. If:*

1. *P is non-trivial: $|P| > 0$ and there exists some Turing machine M such that $\langle M \rangle \notin P$.*
2. *P is a property about the language of the Turing machine: if $L(M_1) = L(M_2)$ then $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$.*

Then P is undecidable.

In this reading, we'll discuss the practical ramifications of this theorem.

Alternative Proof Technique for Undecidability

We can use Rice's Theorem to directly prove that a language is undecidable by showing that the language fulfills the two conditions of the theorem.

Claim 1. *$E_{\mathsf{TM}} = \{ \langle M \rangle \mid M$ is a TM and $L(M) = \emptyset \}$ is undecidable.*

Proof. By Rice's theorem, we must show that E_{TM} satisfies two properties:

1. Non-trivial.
   A TM where there is no path from the start state to the accept state is in E_{TM}, so $|E_{TM}| > 0$.

   A TM whose initial state transitions on a blank to its accept state is not in E_{TM} (its language is $\{ \epsilon \}$), so there exists a TM not in E_{TM}.
2. Language property.
   Suppose that we have two Turing machines M_1 and M_2 such that $L(M_1) = L(M_2)$. We only need to prove the left-to-right ($\Longrightarrow$) direction of the biconditional since the right-to-left direction is symmetric.

   ($\Longrightarrow$): Suppose that $\langle M_1 \rangle \in E_{\mathsf{TM}}$. Thus $L(M_1) =$ and by transitivity, $L(M_2) = \emptyset$ and thus $M_2 \in E_{\mathsf{TM}}$ as well. □

Note that Rice's Theorem only applies to non-trivial properties. For trivial languages, a deciding TM can always be constructed for such properties that either accepts or rejects every input Turing machine.

It is also important to note that Rice's Theorem concerns *language properties* and not *machine properties*. A language property of a Turing machine is a function of the language of the machine. A machine property, in contrast, is a function of the behavior of the machine. For example:

$$L = \{ \langle M \rangle \mid M \text{ is a TM and } M \text{ writes a 0 to the tape during execution on some input} \}$$

is a machine property. For example, two machines M_1 and M_2 could write a 0 to the tape on input 001, but M_1 may accept 001 whereas M_2 rejects it. While this property is undecidable, Rice's Theorem does not apply to it.

Practical Realities

Even if we don't go off to study the theory of computation in graduate school, Rice's Theorem has several important implications for us as software developers. Recall that we postulate that Turing machines are capable of expressing any computation possible by a (classical) computer. Dually, we say that a programming language is Turing-complete if it is capable of expressing any computation that a computer could conceivably perform. Most programming languages that we think of are Turing-complete, including many things—language features, games, and hardware—that we wouldn't think are capable of such expressiveness; the Wikipedia article on Turing-completeness(`https://en.wikipedia.org/wiki/Turing_completeness`) gives several examples.

Rice's Theorem tells us that because Turing-complete programming languages are equivalent in expressiveness to Turing machines (with a caveat that we discuss shortly), then analysis of non-trivial properties of program behavior written in these languages is undecidable. A popular example of this is $\mathsf{HALT}_{\mathsf{TM}}$, the halting problem for Turing machines which we know is undecidable. Because of this, we know that the halting problem for Python or Java is also undecidable. In other words, we can't write an algorithm to detect whether an arbitrary Python or Java program halts on an arbitrary input.

While most developers are not writing program analysis tools, they *use* development tools that are, by the nature of undecidability, at the boundaries of the limits of computation! This has several important implications that we'll now explore briefly. The first is the Full Employment Theorem for Compiler Writers (`https://en.wikipedia.org/wiki/Full_employment_theorem`).

Theorem 2. *(Full Employment Theorem for Compiler Writers) There cannot exist a compiler that fully optimizes all programs in a Turing-complete programming language.*

Proof. Suppose for the sake of contradiction that there exists a fully optimizing compiler for a Turing-complete programming language, say Java. Consider a Java function 'f' that performs arbitrary computation. If 'f' goes into an infinite loop, then such a fully optimizing compiler would optimize 'f' to the follow minimized form:

```
public static void f() {
while(true) { }
}
```

Such a compiler could then be used to detect whether 'f' goes into an infinite loop: compile the program and inspect the optimized output to see if it is in the minimized form. □

The full employment theorem is called as such because a compiler writer's job is never done. A consequence of this theorem is that for any optimizing compiler, there must always be some program that it cannot fully optimize. Therefore a compiler writer can always improve their compiler (although it may not be practical to do so)! As users of compilers, we should realize that while modern compilers are powerful, we can't expect perfection from them in all cases; there will always be room for manual elbow grease when optimizing our programs.

Another consequence of Rice's Theorem is that our development tools have both theoretical and practice limits. For example. Consider the following Java code:

```
public static void g() {
```

```
int x;
// stuff happens...
System.out.println(x);
}
```

We would like the compiler to warn us when we have such *uninitialized variables*. And we might have an intuition about how to proceed: from the definition of 'x', ensure that is has been assigned to at least once before it is used. So we would need to check the '// stuff happens...' block to ensure that 'x' has been given a value.

However, the halting problem can be easily solved if we had such a procedure to check whether a variable is initialized. For example, what if the commented block is the code:

```
while (/* some arbitrary condition */) { }    // potentially an infinite loop
x = 5;
```

Then 'x' is initialized if and only if the while-loop is not an infinite loop. (Also, we never hit the 'println' if the while-loop is infinite, but that's not our concern for the purposes of analyzing whether 'x' is initialized.)

So checking for uninitialized variables is an undecidable problem. Nevertheless, the Java compiler is seemingly able to detect uninitialized variables! The following code:

```
public static void f() {
int x;
System.out.println(x);
}
```

produces the following error:

```
$ javac T.java
T.java:4: error: variable x might not have been initialized
System.out.println(x);
^
1 error
```

What can we infer about the situation given that we know the problem is theoretically undecidable? It must be the case that either (a) Java is not a Turing-complete language so it is possible to decide this property for all possible input programs or (b) an approximation is happening. Indeed (b) is the case that is happening here! For example consider the following Java program:

```
public static boolean alwaysTrue() {
return true;
}

public static void f() {
int x;
if (alwaysTrue()) { x = 5; }
System.out.println(x);
}
```

Clearly 'alwaysTrue' works as advertised, so we always go into the conditional and initialize 'x'. Nevertheless, this code produces the same error as before:

```
$ javac T.java
T.java:10: error: variable x might not have been initialized
System.out.println(x);
^
1 error
```

In practice, static analyses of code are necessarily *conservative* as a result of Rice's Theorem.

- If a piece of code definitely has an uninitialized variable, the Java compiler will definitely report so.
- However, code that may not have uninitialized variables may be flagged by the compiler as having such errors.

As a result, many static analysis tools produce false negatives - cases where a property is not violated but the tool says that the property is violated. This doesn't mean that we should not trust static analysis tools! As developers, we need to recognize that these problems are an inevitable result of having to navigate around the limitations of computation. We may have to rewrite our code to remove such false negatives, or we will need to be willing tolerate this "noise" in our compilation processes without otherwise ignoring the usefulness of the tool.

Acknowledgments

This reading was written by Dr. Peter-Michael Osera. The only changes made were in regard to spacing, and in notation to match *Introduction to the Theory of Computation*, by Sipser, 3rd edition.