12th LogicComputation_Week 3_Lesson 3: Implications of the Halting Problem Name _____
key 1 Per. _____ Date _____

Lesson Overview

Yesterday we saw Turing's proof of (or rather, against) the Halting Problem. We now know that it is impossible to write such a method. (Such a problem is called “undecidable”.)

This should seem a bit surprising. Scanning a file and determining if a program halts could be a very useful tool for a compiler (the tool that optimizes our Java code and turns it into 1s and 0s that the computer can run), and it seems like a reasonable task. Why can we not simply look for loops and recursion and decide whether the code gets “caught”? In the end, though, it is not possible. And we have a proof for that.
It turns out that this has tremendous ramifications for (1) computer science, (2) mathematics, and (3) the way in which we think about the human mind.

Computer Science
There are several problems that turn out to also be undecidable. These turn out to be related to the Halting Problem, but that is often hard to see. The more famous ones are:

1. Given two programs, will they always produce the same output given the same input?
2. Will a particular piece of code ever get executed (or will it be skipped because of looping, etc.)?
3. Is a variable initialized before it is used?
4. Will a variable ever be referenced again? (This is important for memory management – we would want to free up memory if a variable is never used again. Remember, Java has automatic garbage collection … curious, don’t you think, given that we just said this is not possible?)
5. If a “virus” is defined as “a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself,” is a particular program a virus?

Four of these have great ramifications for what a program like BlueJ *cannot* do to help you as a programmer. All of these would be highly useful for checking program correctness. You may notice that BlueJ *does* seem to accomplish some of these. For example, you may have received an error that says a variable was not initialized. Does that contradict the assertion that such a task is not possible? No. Remember, just as the Halting Problem can be solved for *some* programs, it is possible in *some* cases to determine that variables are used prior to being initialized.

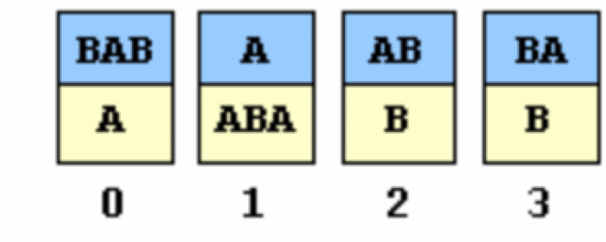
The undecidability of the Halting Problem says that a program cannot determine the case of halting for *all* programs. Likewise, while a compiler can catch *some* uninitialized variables, it cannot find the error in *all* cases. So, compilers do one of two things. Most will catch the ones they can. Some will find cases where they “think” that a variable is being used prior to initialization and generate a warning rather than an error.

The same explanation (frighteningly enough) applies to anti-virus software (#5). It does its best (and a pretty good job), but a fool-proof recognition of a virus is not possible. The way that anti-virus software generally works is by writing code to look for signs of already known about computer viruses and how they work.

The software can search for where virus software may copy itself or ways it might try to communicate with whoever wrote it, or other signatures of tell-tale behavior. But if a new virus is made that is different than any current one, the anti-virus software will be temporarily unable to cope and won’t detect it until the code is updated to handle that new computer virus.

There are also more interesting problems that cannot be decided. They may not have the practical ramifications, but it is so curious that these problems *cannot* be solved. They seem like the kind of thing that *should* be able to be solve.

6. Post’s correspondence problem.
The input to the problem is N different card types. Each card type contains two strings, one on the top, one on the bottom. The puzzle is to arrange the cards so that the top and bottom strings are the same (a yes instance) or to report that it's impossible (a no instance). You can use more than one copy of each card if you want. Here's a yes instance of the problem with 4 card types:



We can solve this by using 1, 3, 0, 2, and a second copy of card 1:

Some card sets do *not* allow for a solution, no matter how hard we try. This set is a "no" set:

The task is to write an algorithm that decides whether a solution can be generated for a given set of cards. The answer: it cannot be done. It is not that we are not good enough. It is not that we have not yet found a solution. It simply cannot be done.

Mathematics

In 1900, David Hilbert addressed the International Congress of Mathematicians in Paris. He posed what he thought the most important unsolved mathematical problems were for the new century (23 of them). His tenth problem was, "Devise an algorithm to determine whether a multi-variable function has an integer root." For example
$f(x,y,z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$

has the integer root (5, 3, 0) because

$f(x,y,z) = 6(5)^3(3)(0)^2 + 3(5)(3)^2 - (5)^3 - 10 = 0$

The task is to write an algorithm to determine whether any given polynomial has an integer root. This Hilbert problem was "solved" finally in the 1970s, in a rather surprising way: it was solved by proving that it cannot be solved. (Note that this problem is decidable if there is only one variable.)

Note that the implications of problems such as these as mean that there are problems that *no* computer system can solve. It doesn't mean that they cannot solve them for certain specific cases, but it does mean that they will never be able to solve all of the problems. It also means, necessarily, that there are some specific problems that a computer system will not be able to handle.

Philosophy

Hilbert's third problem is the one most relevant for us. Hilbert wanted to know if we could develop an algorithm (a process) that would take in a logical statement and a series of formal logical axioms (think here "postulates" and determine if the statement was "true", meaning it could be derived from the axioms. This problem was (is!) essentially important for mathematics. It effectively asks if a computer, by being given axioms, can methodically put them together in order to discover and essentially prove theorems.

Imagine taking the postulates of real numbers, e.g. the Commutative Property of Addition, and turning them into String manipulations: "Whenever you see the string "a+b", you can replace it with the string "b+a". In theory, we could set the computer the applying the rules of manipulation to prove things about numbers. So the task becomes, "Given a statement, very formalized, about numbers, is it a true theorem that can be derived/proven from the axioms?"

This is not possible. In fact, it was Turing's proof of the Halting Problem that definitively showed that it cannot be done. (What the Halting Problem did was to provide at least one theorem than cannot be proved or disproved, but the connection there is difficult and a bit hard to see.)

A computer can never serve as a universal "prover" of algebraic properties.

Nor can it do the same for Geometry properties (think here about a computer generating Euclid's *Elements*).

Nor can it even do the same for formal logic (well, logic robust enough to handle concepts like “for all” and “there exists”).

The human mind can think about these sorts of things. The mathematician Kurt Gödel was working on similar ideas in mathematics (as Turing was in computer science) and came to similar conclusions. Gödel was completely convinced that this means that the “mind” is more than the “matter” (i.e., the brain). He left convinced that the human mind is not a Turing Machine.

No questions to turn in today.

Tomorrow we will wrap up with a video and lecture about Kurt Gödel and his theorem in logic "The Incompleteness Theorem" that is very similar to the Halting Problem.

12th LogicComputation_Week 3_Lesson 3: Implications of the Halting Problem **ANSWER KEY 1**
key 1

Lesson Overview

Yesterday we saw Turing's proof of (or rather, against) the Halting Problem. We now know that it is impossible to write such a method. (Such a problem is called “undecidable”.)

This should seem a bit surprising. Scanning a file and determining if a program halts could be a very useful tool for a compiler (the tool that optimizes our Java code and turns it into 1s and 0s that the computer can run), and it seems like a reasonable task. Why can we not simply look for loops and recursion and decide whether the code gets “caught”? In the end, though, it is not possible. And we have a proof for that.
It turns out that this has tremendous ramifications for (1) computer science, (2) mathematics, and (3) the way in which we think about the human mind.

Computer Science
There are several problems that turn out to also be undecidable. These turn out to be related to the Halting Problem, but that is often hard to see. The more famous ones are:

1. Given two programs, will they always produce the same output given the same input?
2. Will a particular piece of code ever get executed (or will it be skipped because of looping, etc.)?
3. Is a variable initialized before it is used?
4. Will a variable ever be referenced again? (This is important for memory management – we would want to free up memory if a variable is never used again. Remember, Java has automatic garbage collection … curious, don’t you think, given that we just said this is not possible?)
5. If a “virus” is defined as “a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself,” is a particular program a virus?

Four of these have great ramifications for what a program like BlueJ *cannot* do to help you as a programmer. All of these would be highly useful for checking program correctness. You may notice that BlueJ *does* seem to accomplish some of these. For example, you may have received an error that says a variable was not initialized. Does that contradict the assertion that such a task is not possible? No. Remember, just as the Halting Problem can be solved for *some* programs, it is possible in *some* cases to determine that variables are used prior to being initialized.

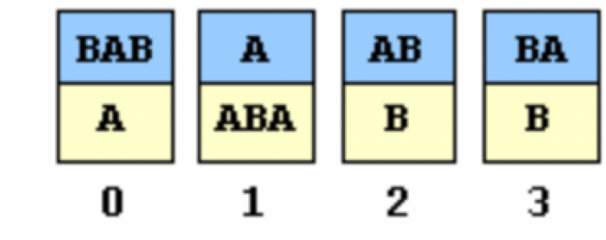
The undecidability of the Halting Problem says that a program cannot determine the case of halting for *all* programs. Likewise, while a compiler can catch *some* uninitialized variables, it cannot find the error in *all* cases. So, compilers do one of two things. Most will catch the ones they can. Some will find cases where they “think” that a variable is being used prior to initialization and generate a warning rather than an error.

The same explanation (frighteningly enough) applies to anti-virus software (#5). It does its best (and a pretty good job), but a fool-proof recognition of a virus is not possible. The way that anti-virus software generally works is by writing code to look for signs of already known about computer viruses and how they work.

The software can search for where virus software may copy itself or ways it might try to communicate with whoever wrote it, or other signatures of tell-tale behavior. But if a new virus is made that is different than any current one, the anti-virus software will be temporarily unable to cope and won’t detect it until the code is updated to handle that new computer virus.

There are also more interesting problems that cannot be decided. They may not have the practical ramifications, but it is so curious that these problems *cannot* be solved. They seem like the kind of thing that *should* be able to be solve.

6. Post’s correspondence problem.
The input to the problem is N different card types. Each card type contains two strings, one on the top, one on the bottom. The puzzle is to arrange the cards so that the top and bottom strings are the same (a yes instance) or to report that it's impossible (a no instance). You can use more than one copy of each card if you want. Here's a yes instance of the problem with 4 card types:



We can solve this by using 1, 3, 0, 2, and a second copy of card 1:

Some card sets do *not* allow for a solution, no matter how hard we try. This set is a "no" set:

The task is to write an algorithm that decides whether a solution can be generated for a given set of cards. The answer: it cannot be done. It is not that we are not good enough. It is not that we have not yet found a solution. It simply cannot be done.

Mathematics

In 1900, David Hilbert addressed the International Congress of Mathematicians in Paris. He posed what he thought the most important unsolved mathematical problems were for the new century (23 of them). His tenth problem was, "Devise an algorithm to determine whether a multi-variable function has an integer root." For example
$f(x,y,z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$

has the integer root (5, 3, 0) because

$f(x,y,z) = 6(5)^3(3)(0)^2 + 3(5)(3)^2 - (5)^3 - 10 = 0$

The task is to write an algorithm to determine whether any given polynomial has an integer root. This Hilbert problem was "solved" finally in the 1970s, in a rather surprising way: it was solved by proving that it cannot be solved. (Note that this problem is decidable if there is only one variable.)

Note that the implications of problems such as these as mean that there are problems that *no* computer system can solve. It doesn't mean that they cannot solve them for certain specific cases, but it does mean that they will never be able to solve all of the problems. It also means, necessarily, that there are some specific problems that a computer system will not be able to handle.

Philosophy

Hilbert's third problem is the one most relevant for us. Hilbert wanted to know if we could develop an algorithm (a process) that would take in a logical statement and a series of formal logical axioms (think here "postulates" and determine if the statement was "true", meaning it could be derived from the axioms. This problem was (is!) essentially important for mathematics. It effectively asks if a computer, by being given axioms, can methodically put them together in order to discover and essentially prove theorems.

Imagine taking the postulates of real numbers, e.g. the Commutative Property of Addition, and turning them into String manipulations: "Whenever you see the string "a+b", you can replace it with the string "b+a". In theory, we could set the computer the applying the rules of manipulation to prove things about numbers. So the task becomes, "Given a statement, very formalized, about numbers, is it a true theorem that can be derived/proven from the axioms?"

This is not possible. In fact, it was Turing's proof of the Halting Problem that definitively showed that it cannot be done. (What the Halting Problem did was to provide at least one theorem than cannot be proved or disproved, but the connection there is difficult and a bit hard to see.)

A computer can never serve as a universal "prover" of algebraic properties.

Nor can it do the same for Geometry properties (think here about a computer generating Euclid's *Elements*).

Nor can it even do the same for formal logic (well, logic robust enough to handle concepts like “for all” and “there exists”).

The human mind can think about these sorts of things. The mathematician Kurt Gödel was working on similar ideas in mathematics (as Turing was in computer science) and came to similar conclusions. Gödel was completely convinced that this means that the “mind” is more than the “matter” (i.e., the brain). He left convinced that the human mind is not a Turing Machine.

No questions to turn in today.

Tomorrow we will wrap up with a video and lecture about Kurt Gödel and his theorem in logic "The Incompleteness Theorem" that is very similar to the Halting Problem.