# Demonstrating the Halting Problem in C#

The Halting Problem plays an important role in the modern understanding of computation. In 1936, Alan Turing famously proved that the Halting Problem is formally ‘undecidable’. It provided a concrete example of the more general theorems of incompleteness published by Kurt Gödel in 1931.

This document provides a brief definition of various terms including formal systems, consistency, completeness, effective procedures, decidability, semi-decidability and the meaning of ‘undecidability’. It introduces the Halting Problem and then provides a description of accompanying C# code that demonstrates the Halting Problem in action.

# Formal Systems

A ‘formal system’ is a system of symbols, axioms and rules. Axioms are a foundational set of propositions that are taken to be true without any proof within the formal system. They provide the basis for all subsequent reasoning within the system, and often represent statements that are regarded as ‘self-evidently’ true (e.g., Φ ∨ ¬Φ, which means “something is either true or false”; this is known as the ‘law of the excluded middle’). Rules include grammatical rules, which determine how symbols can be meaningfully combined, and rules of inference which determine how to derive conclusions from grammatically correct premises.

Formal systems must contain a finite set of symbols. Grammatical rules define the valid ways in which these symbols can be combined to represent formulae. A formal system is said to be ‘recursive’ if there exists some algorithm (an ‘effective procedure’) that can be followed mechanistically or by rote to determine within a finite number of steps whether or not a given formula is a valid axiom or conclusion to a rule of inference within that system – i.e., that the rules of the formal system have been correctly applied.

All recursive formal systems support semi-decidability. Any formal system that supports semi-decidability is described as ‘recursive enumerable’. For semi-decidability, it is sufficient for an effective procedure to halt (complete its work in a finite number of steps) only if a formula is determined to be valid within the given system. If the effective procedure establishes invalidity, or fails to establish validity, it may run forever (i.e., never arrive at a conclusion).

Formal systems are used to reason using symbolic logic and arithmetic. They are also central to computing. Recursiveness, in this sense, lies at the heart of what we mean by ‘computable’. A programming language such as C# provides a general-purpose approach to defining recursive and recursive-enumerable formal systems and implementing effective procedures. If an effective procedure can always determine whether or not formulae are valid within the formal system, we say that this problem addressed by the effective procedure is ‘decidable’. If it can only determine validity in some cases, and never comes to a halt (never completes its work) in others, we say that the problem addressed by the effective procedure is ‘semi-decidable’. If no effective procedure exists within a formal system to determine the validity of a formula, we say that the problem represented that formula is *undecidable*. Any formal system that allows the definition of formulae whose validity is undecidable is said to be ‘incomplete’.

Gödel’s first theorem of incompleteness proves that any axiomatic, recursive formal system capable of representing, at a minimum, basic arithmetic (e.g., Robinson Arithmetic) cannot be both logically consistent (i.e., devoid of contradictions) and complete. His second theorem extends the first by proving that no formal system can prove its own consistency.

# The Halting Problem

The Halting Problem can be stated as follows:

“Is it theoretically possible to create an algorithm (an effective procedure) to decide, for every possible computation over a natural number (0, 1, 2, 3, 4, 5… etc.), if that computation *never* halts?”

Stated in this way, we can see that this problem is semi-decidable. We can consider an effective procedure that halts if it can determine, correctly, that a given computation never halts. This procedure can continue forever otherwise. If the effective procedure continues forever, all we can state is that the computation either halts or that it is not possible to determine if it halts or not. We can’t determine which of these possibilities hold true.

Treating a problem as semi-decidable may seem strange. In programming terms it means that our code may never halt if it cannot establish that some proposition is true. From a practical viewpoint, the last thing a programmer would normally chose to do is to implement a decision-making procedure that may never halt! However, semi-decidability aids our real purpose, which is to illustrate the limits of computability. It simplifies the issue to its ‘bare bones’. The Halting Problem is an entirely legitimate problem in computing. If we can find a computation that we know never halts, and then show that it is impossible to create an effective procedure that determines this by halting, we will have found a formally undecidable problem that demonstrates a limit to computability.

This is the thrust of Gödel’s incompleteness theorems. When applied to computing systems, they imply that for any logically sound programming language (a language that has the property of being ‘Turing complete’), there must exist decision-based problems that we can legitimately and correctly represent within that language, but which no code written in that language could ever determine. Turing showed that the Halting Problem is one such problem. Given that the language is ‘Turing-complete’, our choice of language is of no consequence here. The result applies to any Turing-complete language.

The Halting Problem provides a concrete example of Gödel’s incompleteness theorems at work. It illustrates the natural limits of computing systems. By ‘concrete’ I don’t necessarily mean that it can be practically demonstrated in an exhaustive fashion. Consider an effective procedure that takes a billion trillion years, and which requires more memory than there are quarks in the universe, to arrive at a conclusion. Writing code for this in C# would a waste of time! In similar vein, how could we be certain that a very long-running procedure will eventually complete, at least in theory, or continue forever?

There are other problems that are insurmountable from a practical perspective. A real-world implementation of an effective procedure for the Halting Problem would be able to test any and every computation over a natural number. There are an infinite number of natural numbers. Turing was researching theoretical ‘Turing Machines’ with tapes of unlimited length that could handle an arbitrary number of possible computations. In addition, let’s consider the practicality of implementing the effective procedure. This procedure would have to implement *every possible approach* for determining if a computation over a natural number never halts. Ensuring that this is the case would be very challenging.

We can’t write some huge effective procedure to test an infinite number of computations over an infinite number of natural numbers on theoretical devices that could never exist in reality. However, we can show, logically, that there must be at least one or more computations over a natural number whose halting behaviour can never be determined. This is precisely what Turing achieved, several years before anyone created an actual working computer.

Let’s explore the logic. We need to state the notions of computations and the effective procedure formally. Let’s use C to represent a computation and A to represent the algorithm that acts as our effective procedure.

Our computations take a natural number as an argument. We will represent such a number as ‘*n*’. Each computation, then, has the following signature:

C(*n*)

We can provide a unique number to identify each computation. We will use the set of natural numbers to do this, where *q* is the number that identifies each individual computation:

C*q*(*n*)

Let’s now consider A. This procedure will be used as we search through the problem space, testing each computation in turn. The problem space consists of every C*q*(*n*), so A needs to be called repeatedly for every combination of *q* and *n*. A, then, has the following signature:

A(*q*, *n*)

Now we have specified all Cs and A, we can reason thus:

Consider when *q* equals *n*.

If A(*n*, *n*) halts, then we know that C*n*(*n*) does not halt.

Consider the following.

A(*n*, *n*) is logically equivalent to A(*n*).

A(*n*) is a computation over a natural number. It is therefore a member of the set of all such computations, i.e., a member of the set of C*q*(*n*). We will use *k* to represent the number of the computation that A(*n*, *n*) is. So we can state the following.

A(*n*, *n*) = C*k*(*n*)

We can now state the following for *q* = *n*:

If C*k*(*n*) halts, then we know that C*n*(*n*) does not halt.

Consider when *k* equals *n*. In this case we can re-state the assertion above as follows.

**If C*k*(*k*) halts, then we know that C*k*(*k*) does not halt.**

This is a contradiction. If C*k*(*k*), which is equal to A(*k*, *k*), were actually to halt, then we would know that it never halts…which it just has! This cannot be supported by a consistent system. Hence, assuming that the logic implemented in our A is sound (which implies logical consistency of the system as a whole), our only option is to infer that C*k*(*k*) never halts. It is the only possible behaviour that C*k*(*k*) could exhibit. It means that our effective procedure, C*k*(*k*), which never halts, cannot determine formally that the computation C*k*(*k*) never halts, even though we *know* it never does. This feels very un-intuitive. Our effective procedure implements every possible computational method of determining if a computation never halts, so somehow we have determined something that our algorithmic effective procedure can never determine.

Let’s remind ourselves of the Halting Problem:

“Is it theoretically possible to create an algorithm to decide, for every possible computation over a natural number (0, 1, 2, 3, 4, 5… etc.), if that computation *never* halts?”

We have proved that the answer is ‘no’. We characterise the Halting Problem as formally ‘undecidable’.

# What does this look like in C#?

We saw earlier that there is no practical way to implement an actual effective procedure that could test every computation over every natural number. The problem space is infinite. However, we could certainly implement code that searches through part of the problem space, and if we did that, we might stumble across an example of a non-halting computation that cannot be determined to be non-halting, even if our effective procedure were to implement every possible approach for determining if a computation is non-halting.

It turns out that we don’t have to actually implement such a procedure. We can merely simulate the logic. This is very useful. It allows us to simulate the non-halting behaviour of that algorithm. Our demonstration code can therefore determine that it is simulating non-halting behaviour and then halt and report this. If our algorithm is simulated then the computations it tests can be simulated as well. Our demonstrator only needs to simulate searching through a subset of the real-world problem space and finding a simulated computation that never halts but cannot be determined to never halt.

You may suspect that all this simulation renders the code uninteresting (‘smoke and mirrors’). After all, we could write some code that simply prints a number of simulated results, one of which reports non-halting of our algorithm for a computation we report to be non-halting. Clearly our demonstration code needs to accomplish rather more than that!

In the example code that accompanies this document, we create a subset of the entire problem space as a dictionary of delegates. The index for each dictionary entry represents *q*. Each computation delegates to a lambda that takes a single unsigned integer, allowing us to test over a subset of all natural numbers.

To simulate the effective procedure, the code implements a method with the following signature:

void DoAssessment(uint computationIndex, uint naturalNumber)

In this method, we test for situations where the two arguments have equal values. In this case, we dispatch to an overload of the DoAssessment method with the following signature:

void DoAssessment(uint naturalNumber)

Both methods do the same thing. They create a string representing A(*q*,*n*) and pass this string, together with the computation index and natural number to an internal method that simulates testing the computation for non-halting behaviour. The only difference in the overload is that it specifically tests to see if the indexed computation in the dictionary is the overload method itself. The overload has a signature that is compatible with computation delegates, and the code initialises the dictionary with a number of computations including the DoAssessment overload. If the indexed computation is the DoAssessment method, the string that the method creates represents C*n*(*n*) rather than A(*n*,*n*). This is valid because, of course, A(*n*,*n*) = C*n*(*n*).

The internal test method invokes a helper method that determines if the computation is known not to halt and prints the result using the string created by one of the overloaded DoAssessment methods. The helper method makes its determination in an entirely simulated fashion. By default, the demonstration code uses modulo 3 over the natural number. For each third natural number passed to each computation, the helper method determines that the computation does not halt.

To aid the demonstration, the helper method enters a loop every time it fails to determine that the computation does not halt. This loop is redundant and is included only to represent the notion of semi-decidability. The second time through the loop, the code detects it is in a loop and breaks, returning ‘false’ to represent the semantics of non-determination of the non-halting behaviour of the computation.

The code is written to initially use the default simulated test in all cases. The DoAssessment overload has been carefully added to the computation dictionary at index 6. Hence the simulated test determines that it does not halt and reports the following:

Computation\_6(6) halts, therefore the program knows that Computation\_6(6) does not halt.

This message, of course, makes no logical sense. Our only way to fix what is clearly a logical bug is to add a special test for Computation\_6(6). This test fails to determine if the computation halts. The test is compiled by un-commenting the AssessorTest symbol at the top of the code. Now, when the code runs, the program report the following:

Computation\_6(6) does not halt, therefore the program does not know if Computation\_6(6) halts.

This message reflects the fact that we know the computation does not halt. The DoAssessment overload, when invoked with the value 6, enters the ‘never-ending’ loop. It is simulating the testing of the DoAssessment overload, as the computation, when invoked with the value 6. The only logical option available to us is to report that the program cannot determine the non-halting behaviour of the computation, even though, as the accessor, the same code reports that it does not halt. The messages deliberately refer to what the program ‘knows’. The reader might wish to reflect on the apparent difference here between the insight available to humans and the knowledge that can be deduced by the programme.

# What does this mean?

We can see that computation, as we understand the concept, has limits. C# is a ‘Turing-Complete’ language running on a practical approximation of a Universal Turing Machine. It is a characteristic of Universal Turing Machines that they are all formally equivalent. Discounting considerations of time (performance) and space (memory), any computational logic that can be executed by one Universal Turing Machine can be executed by all Universal Turing Machines.

Gödel’s incompleteness theorems indicate that there is no way out of the conundrum represented by the Halting Problem. Very simply, there are some problems that are not formally decidable through computation, regardless of the sophistication of the machine or, indeed, the programming language.

It is very tempting to make additional deductions of a more philosophical nature. Gödel’s incompleteness theorems have been repeatedly abused in this way, even to the point of supposedly providing ‘proofs’ of both the existence and non-existence of God, based on an entirely fallacious inference from the theorems on the limits of human knowledge. Another controversial, but possibly sound idea is that the incompleteness theorems, and hence the Halting Problem, illustrate some ability of the human mind to attain mathematical insights that are not available via any form of computation, and which are therefore inaccessible to any computer. This claim is by no means widely accepted, but the argument has been made cogently and repeatedly. The best-known advocate of this idea is Sir Roger Penrose.

If Penrose and others are correct, it will not prove possible to create a ‘strong’ artificial intelligence that exhibits conscious awareness on the basis, solely, of computation. This is often regarded as equivalent to claiming the impossibility of an artificial *general* intelligence arising from Turing machines (or rather, their approximations in the real world – e.g. computers). Unfortunately, science has yet to ascertain the mechanisms of consciousness. We do not even know with any certainty how general anaesthetics temporarily erase our conscious mind. It is not possible currently to prove or disprove any of the numerous claims and conjectures with regard to conscious awareness. We perhaps cannot even determine how a claimed artificial general intelligence, should one arise, could be deemed ‘the real thing’.

Formal undecidability, as encountered in the Halting Problem, is a fundamental, but bewildering, limitation in axiomatic logic, arithmetic and computing. Its true meaning remains a matter of controversy. It relevance to philosophical enquiry is unclear. Its implications for the future of machine-based reasoning are fiercely debated. I have no answers to these questions, but I hope this small contribution to the subject will help programmers to better understand the nature of incompleteness.

**Charles Young**December 2015

**Scooping the Loop Snooper**

an elementary proof of the undecidability of the halting problem

Geoffrey K. Pullum, University of Edinburgh

No program can say what another will do.   
Now, I won’t just assert that, I’ll prove it to you:  
I will prove that although you might work ‘til you drop,  
you can’t predict whether a program will stop.

Imagine we have a procedure called P  
that will snoop in the source code of programs to see  
there aren’t infinite loops that go round and around;  
and P prints the word “Fine!” if no looping is found.

You feed in your code, and the input it needs,  
and then P takes them both and it studies and reads  
and computes whether things will all end as they should  
(as opposed to going loopy the way that they could).

Well, the truth is that P cannot possibly be,  
because if you wrote it and gave it to me,  
I could use it to set up a logical bind  
that would shatter your reason and scramble your mind.

Here’s the trick I would use – and it’s simple to do.  
I’d define a procedure – we’ll name the thing Q –  
that would take any program and call P (of course!)  
to tell if it looped, by reading the source;

And if so, Q would simply print “Loop!” and then stop;  
but if no, Q would go right back to the top,   
and start off again, looping endlessly back,  
‘til the universe dies and is frozen and black.

And this program called Q wouldn’t stay on the shelf;  
I would run it, and (fiendishly) feed it *itself*.  
What behaviour results when I do this with Q?  
When it reads its own source, just what will it do?

If P warns of loops, Q will print “Loop!” and quit;  
yet P is supposed to speak truly of it.  
So if Q’s going to quit, then P should say, “Fine!” –  
which will make Q go back to its very first line!

No matter what P would have done, Q will scoop it:  
Q uses P’s output to make P look stupid.  
If P gets things right then it lies in its tooth;  
and if it speaks falsely, it’s telling the truth!

I’ve created a paradox, neat as can be –  
and simply by using your putative P.  
When you assumed P you stepped into a snare;  
Your assumptions have led you right into my lair.

So, how to escape from this logical mess?  
I don’t have to tell you; I’m sure you can guess.  
By *reductio*, there cannot possibly be  
a procedure that acts like the mythical P.

You can never discover mechanical means  
for predicting the acts of computing machines.  
It’s something that cannot be done. So we users  
must find our own bugs; our computers are losers!