Chapter 1

A Game We Can't Win: Undecidability of Quantificational Logic

Jonas Wechsler, Mehul Bhatnagar, Sam Grayson

"I know that I know nothing."
- Socrates

General Todos: italicize the first instance of vocabulary words, format graphics better, refer to graphics as fig n, format theorems and defns, spellcheck.

1.1 Computers

1.1.1 Alan Turing's life

Alan Turing (born June 23, 1912 in London) demonstrated intelligence at an early age first at St. Micheal's then at Sherborne School. After grade school, Turing studied and subsequently taught mathematics at Cambridge University. At Cambridge, he wrote "On Computable Numbers, with an Application to the Entscheidungsproblem", a paper on the limits of proof and computation that created formal and simple hypothetical devices that ultimately became known as Turing Machines.

During World War II, he worked for the British to crack German codes at Bletchley Park. There he developed an early computer that could guess the key to decrypt the Enigma machine, providing the Allies with crucial information. He made these computers practical and had other contributions to cryptography for the British. He was known by his coworkers for his quirky habits.

His work did not go unnoticed. In 1945, Turing was awarded the Most Excellent Order of the British Empire (OBE) for his help, though the projects remained secret for many years.

After the war, he put more research into computers at the National Physics Laboratory (1945-1948). Then at Manchester University he published "Computing machinery and intelligence", which formed a foundation for artificial intelligence and formulated the Turing test, a test of functional AI.

In 1951, three years before his death, he was elected a fellow of the Royal Society. In 1952, Turing was arrested for homosexuality. When given a choice between prison and chemical castration (estrogen shots), he chose the latter. At the time, homosexuals were considered to be at risk of blackmail and therefore a security risk. His security clearance was denied and he was no longer able to work at GCHQ, the post-war successor to Bletchley Park.

He committed suicide on June 7, 1954.

1.1.2 Turing Machines' Formal definition

To more easily analyze the abilities of computation, Turing proposed a theoretical computer, consisting of a head that moved back and forth along a strip of tape. This machine was known as a Turing Machine. The tape was infinitely long and divided into square cells (like movie reel tape). Each cell had a single unit of data written on it. The head moved up and down the tape, reading and writing to the tape depending on a set of rules.

The head has an internal state or a set of rules that describe how the head operates. Every cycle, it reads the current cell, and based on the state decides what to write to the current cell, whether to move the tape to the right or move to the left, and what new state to go to. There is a special state called 'halt' which indicates the machine is done executing the program.

A program for the turing machine consists of an alphabet of data that can be written to the tape, a set of states which describe how the head operates, and a tape with default values on it. There is a special symbol for the tape which means 'blank' and a set of states which mean 'halt'. The rest is up to the programmer.

1.1.3 Example Turing Program: Incrementing and decrementing

Let our alphabet be $\{0, 1, A\}$, where 0 denotes the blank. Let us use a Turing machine to increment a number by one. We can set up the tape such that the number we want to increment is written on the tape in binary. We can move the machine such that the head is over the left-most digit (least-significant digit) of the number and an A denotes the right-most boundary. The tape extends *ad infinitum* to the left and to the right.

In binary addition, $0+0=0$, $1+0=1$, $0+1=1$, $1+1=10$ (zero carry one). To carry one, move to the left and add one again.

$$\begin{array}{ccccc}
 & 1 & 1 & & \\
 & 0 & 1 & 1 & 1 \\
+ & 0 & 1 & 1 & 0 \\
\hline
 & 1 & 1 & 0 & 1
\end{array}$$

If that bit is a 0, to increment it we change it to a 1. If the bit is a 1, we set it to 0 have to carry one. Notice that to carry one, we just add one to the second lowest significant bit (place value of two), move to the left, and increment from that position. If we have to carry so much that we hit the left delimiter A, that means the number is too big to fit in the space provided. This is called an integer overflow. I want the program to crash.

Current state	Symbol read	Symbol wrote	Movement	Next state
Increment	0	1	No movement	Halt
	1	0	Left	Increment
	A	A	No movement	Crash (overflow)

The Turing Machine's head is currently over the red cell. It is set up to increment 3 (011). It will execute, step by step until 'halt' is reached.

$\cdots$	A	0	1	1	$\cdots$	
$\cdots$	A	0	1	$1 \to 0$	$\cdots$	Increment $\to$ Increment, left
$\cdots$	A	0	$1 \to 0$	0	$\cdots$	Increment $\to$ Increment, left
$\cdots$	A	$0 \to 1$	0	0	$\cdots$	Increment $\to$ Halt, no movement
$\cdots$	A	1	0	0	$\cdots$	Halted

The tape below is set up to increment 3 (011). It adds one getting 4 (100), carrying twice. Decrementing works the same way, except the 1 and 0 are switched, so $0 \to 1$, decrement $\to$ decrement (carry one), and $1 \to 0$, decrement $\to$ halt.

Current state	Symbol read	Symbol wrote	Movement	Next state
Decrement	1	0	No movement	Halt
	0	1	Left	Increment
	A	A	No movement	Crash (underflow)

Not that when we try to decrement from the tape $A0000$, we find 0 as the current symbol, change it to 1, move to the left, and repeat (carry one). Then we find 0 as the current symbol, change it to a 1, move to the left, and repeat until we reach A which tells us that the integer underflowed.

1.1.4 Example Turing Program: Adding Natural Numbers

Let our alphabet be the $0, 1, A, B, C, D$, where 0 denotes a blank. We want to put the first number on the tape towards the left, and then the second number towards the right. In order for the Turing Machine to know where the numbers start and end, I will put an A to denote the start of the first number, B for its end, C for the start of the second number, and D for its end. Let the turing machine start anywhere to the right of B.

The following diagram is an example of the tap for this program set to compute 5 + 6. The 5 is between A and B. The 6 is between C and D. There can be an arbitrary amount of data between B and C; It is all ignored by the program. This will become important later.

···	0	A	0	1	0	1	B	···	C	1	1	0	D	0	···

In general want to decrement the first number and increment the second number until the first number is zero. Then we know the sum is stored where the second number used to be second number (between C and D).

1. In the style of the first example, I will navigate to the start of the first number (denoted by B) using the state 'rewind'. This state moves to the left until it hits a B, and then moves exactly once more to the left.
2. Using the second example, I will decrement the first number by one under the state 'decrement'. If the there is an underflow, that means I tried to decrement from zero, therefore the machine is done adding and it should halt.
3. Otherwise I want the machine to move back to the start of the second number (deonted by D) under state 'forward'.
4. Now that the head is positioned, I want to increment the second number by one using the 'increment' state. If there is an integer overflow, the second block of memory is not large enough to store the result, so the machine crashes. Otherwise repeat.

Current state	Symbol read	Symbol wrote	Movement	Next state
Rewind	B	B	Left	Decrement
	Otherwise	*None*	Left*	Rewind
Decrement	1	0	Right	Forward
	0	1	Left	Decrement
	A	A	None	Halt
Forward	D	D	Left	Increment
	Otherwise	*None*	Right*	Forward
Increment	0	1	Left	Rewind
	1	0	Left	Increment
	C	C	None	Crash

Note that the program sets the first addend, AB, (where xy means the number between x and y) to zero while it sets the second addend CD to the sum. If we want to reverse these roles, we simply change 'left*' to 'right' and 'right*' to 'left' (the asterisk is ignored by the Turing Machine; it used as a marker for educational purposes).

1.1.5 Example Turing Program: Multiplying Natural Numbers

Multiplication is repeated addition. The adder we built previously decrements its first input until it is zero, while the second input stores the result. In order to add numbers multiple times, we will need an adder that doesn't destroy first input. The second input will act as an accumulator for the sum, so it doesn't have to be reset.

In order to construct multiplication, we need another memory block delimited by EF which starts out at zero. We will decrement the number in AB, then increment the number in CD (like before), but then we will also increment the number in EF before repeating. In the interest of time, the implementation will not be shown here. Instead, it is summarized below.

1. Decrement AB, go to step 5 on underflow.
2. Increment CD, crash on overflow.
3. Increment EF, crash on overflow
4. Repeat
   (when loop terminates, $-1 \to AB$, $AB + CD \to CD$ $AB \to EF$)
5. Decrement EF, go to step 8 on underflow
6. Increment AB, crash on overflow
7. Repeat
   (when loop terminates, $AB \to AB$, $AB + CD \to CD$, $-1 \to EF$)

8. Zero EF
   (when program halts, $AB \to AB$, $AB + CD \to CD$, $0 \to EF$)

Zero EF is a state that starts at F, if E is found halt, otherwise write zero, move to the left and repeat. Note that -1 denotes the number you get when you decrement one from zero (the number with a 1 at every digit).

Notice that the 'add, preserve first input' program can be used to copy one block of memory (AB) to another place (CD) provided that the other place is zero, using a temporary buffer (EF). This is because $AB \to AB$ so it remains unmodified. $AB + CD \to CD$, but $CD = 0$, so $AB \to CD$. Thus we can copy AB to CD. By changing the direction of the adding we can even copy CD to AB.

Now we have the tools to calculate $m \cdot n$. Let the $0 \to AB$, $m \to CD$, $n \to EF$, and $0 \to GH$.

AB is used as a counter to tell us how many times to add CD to EF. When we add CD to EF, we use 'add, preserve first input' such that CD is unchanged and EF accumulates the result while GH is used as a temporary buffer. The algorithm is summarized below.

1. Copy CD to AB
2. Decrement AB by one, halt on underflow.
3. Add, preserving the first input, CD to EF using GH as a temporary buffer, storing the result in CD (remember that this program needs a temporary storage buffer). Crash on overflow.
4. Repeat, starting from step 2.
   (when the program halts, $-1 \to AB$, $CD \to CD$, $CD \cdot EF \to EF$, $0 \to GH$)

1.1.6 Example Turing Program: Computing Natural exponents

We want to calculate m^n for two natural numbers m and n. Let the tape be set up where $n \to AB$, $0 \to CD$, $m \to EF$, $1 \to GH$, and $0 \to IJ$. We will decrement AB until underflow, each time multiplying EF and GH.

1. Decrement AB, halt on underflow
2. Multiply using CD (storage), EF (m), GH (n), IJ (storage), crash on overflow (Remember, $-1 \to CD$, $EF \to EF$, $EF \cdot GH \to GH$, $0 \to IJ$; Notice that GH is used to store the second factor and the result is written to GH, so during the next iteration, the result of the previous iteration is multiplied by EF)

3. Repeat.

1.1.7 The Halting Problem

Through out all of these programs there is recursion (meaning a repeating loop that calls itself). For example consider the, "Search for character" program

Current state	Symbol read	Symbol wrote	Movement	Next state
Rewind	B	B	No movement	Halt
	0	0	Left	Rewind
	1	1	Left	Rewind

What happens if the head starts out to the left of the A instead of to the right? What happens if there is no A on the tape? In these cases the Turing Machine goes into an infinite loop.

There is in fact no way to tell if any program will halt in finite time. Sometimes, like in the cases stated above with the "Search for character" program, we know that the program is going to be in an infinite loop if A does not occur, but you could imagine a more difficult program that has thousands of states. While we do know that some programs won't halt given certain conditions, there are some programs for which we don't know if it will halt.

1.2 Axiomatic systems

1.2.1 What is an axiomatic system?

You should remember first-order logic has names (also called constants) denoted by lower case letters a, b, c. It also has predicates denoted as Pxy or $P(x, y)$ (where the predicate P is acting over the arguments x and y) that evaluate too booleans. Thus Px is usually true or false. There are also functions denoted the same way as predicates that evaluate to objects, so $f(x)$ is not true or false, but rather some object like x or 2 for example. There are also logical connectives like $\wedge$, and most importantly the quantifers $\exists$ and $\forall$.

An axiomatic system is a set of sentences in quantificational logic that describe something. These are very useful in mathematics. Natural numbers can be defined as an axiomatic system.

1.2.2 Example Axiomatic System: Natural Numbers

Let Nx mean x is a natural number, and Sx mean the next number after x (also called successor of x).

1. $\neg\exists x(Sx = 0)$ There is not a number whose successor is zero.
2. $\forall x\forall y(Sx = Sy \rightarrow x = y)$ Equal successors ($Sx = Sy$) mean equal numbers $x = y$.
3. $\forall y(\neg y = 0 \rightarrow (\exists x Sx = y))$ Except for zero, every number y is the successor of some number x.
4. $\forall x(x + 0 = x)$ Identity property of addition.
5. $\forall x\forall y(x + S(y) = S(x + y))$ Defines addition recursively (together with the identity property)
6. $\forall x(x \cdot 0 = 0)$ Zero property of multiplication
7. $\forall x\forall y(x \cdot Sy = x \cdot y + x)$ Defines multiplication recursively (together with the identity property)
8. $(P(0) \wedge \forall k(P(k) \rightarrow P(k + 1))) \rightarrow \forall n P(n)$ A statement of induction

This is called *Peano Arithmetic*. There are many different versions, here is a common one. It encodes natural numbers in first-order logic.

1.2.3 Models

You should also remember that we can specify an *interpretation* which is just an example of objects that satisfy make all of the sentences true. We specified models by creating a *model*, which states the universe of discourse and specifying when predicates are true.

The Natural numbers you know are a model for Peano Arithmetic. $UD = \{0, 1, 2, \ldots\}$ where $S = \{(0, 1), (1, 2), \ldots\}$

1.2.4 Completness and Consistency

A formal system consists of syntax and semantics. Syntax tells us how to use the symbols to produce statements. Semantics tells us how to use the statements to deduce other statements. While syntax tells us how the language ‘sounds’, semantics tells us what language ‘means’.

A formal system is complete if and only if every statement can be proven either true or false. A formal system is consistent if and only if no statement can be proven true and false.

1.2.5 Entscheidungsproblem

David Hilbert proposed several famous questions. One of them asks for a procedure to determine if a sentence in first-order logic was valid (recall valid means true for every case that satisfies the premises). It is calling for a why to prove things that always works.

We studied Natural Deduction as a way to prove things in first-order logic. This sometimes works. Consider a invalid argument. Natural deduction will deduce things that are true, but when using natural deduction there are some arguments that can't be shown to be valid or invalid.

Consider the following argument which can neither be proven true nor proven false using natural deduction.

$$\begin{array}{l} \exists x Px \\ \exists y Py \\ \hline \exists x(Px \land Py) \end{array}$$

With these premises, we can never deduce $\exists x(Px \land Py)$ or $\neg\exists x(Px \land Py)$. Natural deduction continues indefinately, not proving this statement.

1.3 Observations

1.3.1 Church-Turing Hypothesis

The Church-Turing thesis proved that three methods of formalizing the idea of computability were all equivalent, and that these methods coincide with the informal notion of an effectively calculatable function. Informally, an effectively calulatable function can be done by a human with a pencil and paper, ignoring resource limitations (including time). The three methods of formalizing computability were:

1. Kurt Gödel and Jacques Herbrand's idea of the General Recursive Function.
2. Alonzo Church's method called the λ-calculus
3. The Turing machine

This is important because it says that if I want to see if my computer is capable of computing any function (barring memory and time constraints), I can see if it can simulate a Turing Machine, or see if I can implement Lambda Calculus, or see if I can encode recursive functions. If any of these are possible, my computer is powerful enough to theoretically calculate anything (within time and memory constraints).

This is most significant because if a problem is unsolvable with a Turing machine, then the problem must be incomputable. No other method could possibly solve it. But what problems could possibly fall into this category?

1.3.2 Gödel's Incompleteness Theorems

Gödel's First Incompletness Theorem Every set of axioms is either inconsistent or incomplete, where incomplete means that it cannot prove all truths about the natural numbers.
Gödel's Second Incompletness Theorem The second incompleteness theorem shows that such a system cannot demonstrate its own consistency.

1.3.3 Informal proof

Lets say we have a set of axioms, S. Assume S is complete. Since S is complete, we can write a computer program, P, and prove "P halts" or "P doesn't halt" using rules from S.

Let P be a program that makes a copy of itself, called R, and uses rules of S to prove "R does not halt." Once it proves this, P halts.

Notice if P halts, it must have proven "R does not halt," but R is a copy of P, therefore P does not halt. Contradiction

If P does not halt, then "R does not halt" must be not provable in finite time. Thus there are statements in S that can't be proven. Therefore S is inconsistent. Contradiction.

Thus Gödel's first Incompletness Theorem is proven.

1.3.4 Incompleteness implications: Entscheidungsproblem

Any proof system in first-order logic is either incomplete or inconsistent, which means that we cannot fully analyze a program, which informally means that we cannot deduce if a statement is universally valid. Gödel's work influenced the work of Turing and Church heavily, but it was not sufficient to rigorously solve the Entscheidungsproblem.

1.3.5 Incompleteness implications: Halting problem

We can't be sure that a given turing program will halt. We know this because the program cannot be fully analyzed, as informally implied by Gödel, so we will not be able to fully deduce the fate of the program. Again, the work of Gödel laid the foundation for future research, but it was insufficient to formally solve the halting problem.

1.3.6 Incompleteness implications: Continued implications

You can't prove *everything* in first-order logic. There will always things which you can not prove.

The laws of physics in our universe can emulate a Turing machine (since we have built real life Turing machines), therefore the laws of physics in our universe is at least as powerful as a Turing machine.

Some claim it is equal in power to a Turing machine (there exists a precise set of rules, even probabilistic rules, to model nature). This implies that every physical situation in the universe is computable. All the laws of physics can be simulated by a computer and described through numbers. This is called digital physics, and it contradicts some of quantum physics, so most people are skeptical.

The other option is that it is greater in power than a Turing machine, in which case could we use physical phenomena to create a computer more powerful than a Turing machine? This idea has been termed the hypercomputer. Still others speculate that althought the universe contains physical phenomena that can not be simulated with a Turing machine, this phenomena is not 'harnessable' for the construction of a hypercomputer.