Diagonalization

Cardinalities

The cardinality of a finite set is easy to grasp: $|\{1,3,4\}| = 3$. But what about infinite sets?

We say that a set S *has at least as great cardinality as set* T, written $|S| \geq |T|$, if there is an onto function f from S to T.

For example, $|\mathbf{R}| \geq |\mathbf{N}|$ there are at least as many reals as there are natural numbers. The onto function from **R** to **N** maps natural numbers to themselves and every real that is not a natural number to 0, say. Similarly, $|\mathbf{Z}| \geq |\mathbf{N}|$ (it's always trivial that the cardinality of a set is at least that of a subset).

But it turns out that also $|\mathbf{N}| \geq |\mathbf{Z}|$, and so we can write $|\mathbf{Z}| = |\mathbf{N}|$. The onto function that establishes this is, for example, $f(2i) = i$, $f(2i+1) = -i$.
We also know from the previous lecture that $|\mathbf{N}| = |\mathbf{Q}| = |\mathbf{N}^2|$ (by Cantor's "sweeping" trick).

How about $|\mathbf{R}|$ and $|\mathbf{R}^2|$, do they also have the same cardinality? The answer is "yes". The onto function from $|\mathbf{R}|$ to $|\mathbf{R}^2|$ is the following (by example):
f(3718.056921…) = (31.062…,78.591…): you separate the even digits from the odd digits.

It is also easy to see that $|\mathbf{R}| = |[0,1]|$ (exercise: find an onto function). Finally,

$$|[0,1]| = |2^{\mathbf{N}}|$$

(remember, 2^S , the *power set* of S, is the set of all subsets of the set S). To see this, just encode both reals and sets of integers as infinite binary strings (an trick is needed to take care of the "double binary encoding" problem, as in .011111… = .100000…).

Recall now the proof that $|\mathbf{N}| < |[0,1]|$ from last time ("<" means "≤ and not ≥" ; that theorem said that there are more real numbers in the interval [0,1] than there are integers). In view of the equation $|[0,1]| = |2^{\mathbf{N}}|$ above, this is a special case of the following general result:

Theorem (Cantor, 1880): For every set S, $|S| < |2^S|$.

Proof: We want to show that there is no onto function from S to 2^S. Suppose that such a function f existed. Then consider the following subset of S called D (for "diagonal")

$$D = \{ \text{ a in S: a is not in f(a)} \}$$

This is an element of 2^S, and so, since f is onto, there is an element of S, call it d, such that f(d) = D.

And here is the unanswerable question, a contradiction establishing that f cannot exist: is d an element of D? It is iff it is not (think about it!).

This is the famous *diagonalization* argument. It can be thought of as defining a "table" (see below for the first few rows and columns) which displays the function f, denoting the set f(a1), for example, by a bit vector, one bit for each element of S, 1 if the element is in f(a1) and 0 otherwise. The diagonal of this table is 0100…. If we flip it we get the set D = 1011… This is a set that is guaranteed not to be a row of the table (because it differs from the i-th row in the i-th bit), proving that f is not onto.

a	f(a)			
a1	0	1	1	0
a2	1	1	1	1
a3	0	1	0	1
a4	0	1	1	0

It follows from Cantor's Theorem that $|\mathbf{N}| < |2^{\mathbf{N}}| = |\mathbf{R}| < |2^{\mathbf{R}}| < \ldots$ an infinite hierarchy of ever increasing cardinalities…

A last question: Is there a cardinality between $|\mathbf{N}|$ and $|\mathbf{R}|$? This is the famous *Continuum Hypothesis*, proposed by Cantor; it is now known that this possibility *and* its negation are both consistent with set theory…

The halting problem

The diagonalization method was invented by Cantor in 1881 to prove the theorem above. It was used again by Gödel in 1931 to prove the famous Incompleteness Theorem (stating that in every mathematical system that is general enough to contain the integers, there must be theorems that have no proofs). And again by Turing in 1937 to establish that there are perfectly reasonable computational problems, which are *unsolvable*: there is no algorithm (program) for solving them.

Let's talk about the latter (and keep the Incompleteness Theorem for last). Suppose that somebody told you that they have written a program with the following specs:

halts (M, I: files)
It returns "true" if the program in file M, when supplied with the data in file I, runs for a finite amount of time and then stops (in other words, it does not loop); it returns "false" if M loops on I.

Would you believe them?

Theorem (Turing, 1937): There is no program that does the above.

Proof: Suppose such a program exists. Then we write the following program (with the ominous name D, from the previous proof):

```
   D(M : file)
1: if halts (M, M) goto 1
```

Notice what D does: If the program in its input file halts when supplied with itself as input, then D halts, otherwise it loops forever. But now suppose that we put program D in a file called D, and then we run D(D). Will this computation halt? It will iff it will not (check!), and this is a contradiction.

The unsolvability of the halting problem is a very important insight for CS. It tells us that there are limits to computation, to algorithms, to software engineering. You cannot tell whether a program will halt – but neither can you tell if a program has a buffer overrun, or is redundant in that lines can be deleted with impunity, etc. etc. It means that programming and debugging and analyzing code will forever remain forms of art and hackery. And what is doubly amazing is that this important CS insight was articulated at the same time that CS was born (because Turing's paper that contained this theorem was the same that defined the Turing machine, the mathematical abstraction of a computer that started us on the way to the von Neumann computer and its software).

The Incompleteness Theorem

Logic is a mathematical field that studies Mathematics itself. We have seen Boolean Logic and First Order Logic at the beginning of the course. The latter, augmented with operations like + and * and ↑ (exponentiation) and relations like < and constants like 0 and 1 (and of course variables and "for all" ∀, there exists" ∃, and Boolean connectives AND, OR etc.) can be used to express very complicated mathematical sentences about the natural numbers. This language is called *First-Order Arithmetic*, or FOA. For example, these are sentences in FOA:

$\forall x\ \forall y\ \exists z\ [z = x + y]$
$\forall x\ \exists y\ [x = y + y \text{ OR } x = y + y + 1]$
$\forall x\ \exists y\ [x = y + y]$
$\forall x\ [x = 1 + 1 + 1]$
$\forall x\ [x = y]$

The first two are theorems (true sentences about the natural numbers), stating that "addition is a fully defined operation" and "every number is either even or odd," respectively, while the next two are false. (The last is not even a sentence, because its truth depends on the value of variable y…)

What would a proof for such a true sentence be? Presumably it would start from some axioms, stating some basic facts about Boolean connectives and properties of the quantifiers, plus some important properties of the natural numbers, like "Ax [0 + x = x]". Then it would allow us to deduce new true sentences from axioms and already established ones, until the sought theorem is produced. We don't need to define proofs precisely; we will just specify that proofs are finite strings that have the following three properties:

1. If P is a proof of statement A, then it can be checked as such. That is, there is a program proves(A,P) that always halts and decides whether P is a valid proof for A.

2. If P is a valid proof for A, then A is a true sentence, a theorem about the natural numbers. That is, our proof system is *sound*, it does not prove false statements.

We can now state the Incompleteness Theorem:

Incompleteness Theorem (Gödel 1931): In any system with these properties, there are (true) theorems in FOA that have no proofs.

Proof: We show that it follows from the unsolvability of the halting problem (which Turing proved, however, having been influenced by Gödel's proof of the IC). It proceeds in these steps:

1. The halting problem is unsolvable even if it is tightened as follows: Given a program *with no input statements*, will it halt? This is done by reducing the original form of the halting problem to this one: Given a program M and a data bitsring I, define a new program M', without input, that does the following: It starts by setting internal variables to the data bitsring I, and then it simulates P by looking up any needed input. Then telling whether M' halts is the same as telling whether M halts on input I.

2. Given such a program M, the statement "program M will halt" can be restated as a sentence S(M) about natural numbers. S(M) is either a theorem (if M indeed halts) or its negation is a theorem. Of course to carry out this step we have to fix a programming language, and here we use the simple programming language called "Turing machine" due to Turing himself. We show how this is done in the Appendix below.

3. Now we are done: If every theorem had a proof, then we could use the "proves" routine to come up with a program that looks through all possible proofs and solves the halting problem – a contradiction. In more detail, given a program M, the following algorithm decides whether M halts or not:

   Start with constructing the statement S(M)
   For i = 0, 1, 2, … do: For each possible proof P of length i do:
   If proves(P,S(M)) then output "M halts" and stop
   If proves (P, NOT S(M)), then output "M loops" and stop

This contradicts the unsolvability of the halting problem, completing the proof of the Incompleteness Theorem.

APPENDIX

Here we show how, given a program M, we can construct a statement in FOA S(M) such that S(M) is a theorem iff M halts.

First we must define a programming language. We choose a very primitive, but still completely general, programming language known as "Turing machine," invented by Turing.

In this language, a program M operates on a data structure consisting of an infinite string. This string is initially all zeros (think of 0 as a blank). Each position in the string contains a character from an alphabet, say 0 … 9 (different programs may have alphabets of different sizes). At each point in time, the program is looking at a particular position in the string, and acts on this position in a way specified by the current *instruction*. The machine M has a number of instructions, say four instructions A, B, C, D. One of these instructions (by convention A) is the *starting* instruction. All instructions are of the same type: Depending on the character at the current position of the string, three things are done: A new character is overwritten; the current position is moved to the right (R), left (L) or is unchanged (U); and a new instruction is executed next. That is, every instruction is a case statement; for example, instruction A could be

A: if current = 0 then {write 2, move R, goto C}
if current = 1 then {write 0, move L, goto A}
if current = 2 then {write 2, move U, goto B}
.
.
.
if current = 9 then {write 7, move L, goto F}

The other instructions of M, instructions B, C, and D, are similar. There is also a special instruction F ("freeze!") whereby the program halts.

Programs in this language may have many more instructions or symbols. The point is that this primitive programming language can simulate any other programming language, like C++! (Does this sound believable? It's true!)

To represent the operation of this program, we make the convention that we omit the infinite 0 portions that are to the left and right of the "active" part of the string, and represent a state of the program by the remaining string (without trailing or initial 0s) with the instruction being executed inserted just to the left of the current position. This is called a configuration. For example, configuration 18A93452 means that the program is about to execute instruction A, that the string is …00…00189345200…00… and that the underlined position is the current one. According to our description of instruction A above, the next time step the configuration will be 1F873452, and the execution will stop.

A computation of the program M above can be denoted as sequences of configurations, separated by → . The first configuration is thus A0 (the current symbol is a 0, surrounded by infinitely many other 0's, and the instruction executed is the start instruction A). From that (according to our example A above) we go to configuration 2C0, and from that perhaps (if it is the case that C contains "if current = 0 then {write 1, move L, goto A}") to A20 and from there to B20, and so on. We represent this as

$$A0 \rightarrow 2C0 \rightarrow A20 \rightarrow B20 \rightarrow \ldots$$

Notice that, if we encode → as E, this becomes a *hexadecimal number!* The bottom line is that sequences of configurations of a program can be thought of as numbers.

We are now ready to construct S(M), the sentence on FOA that says "program M halts." S(M) is of the form

$$S(M) = \exists x \, [\, B(x) \text{ AND } W(x) \text{ AND } H(x)],$$

stating that there is a number x which encodes a halting computation of machine M. The three conjuncts B(x), W(x) and H(x) are themselves FOA statements declaring that computation x starts right (its first three symbols are A0→); that its works correctly (the parts between any three consecutive →'s follow from one another according to the instructions of the program), and that it halts (the symbol F appears somewhere).

Here is how you write these parts:

B(x) is the following: $\exists z \, [\, x \text{ div } 16 \uparrow z = (A0E)_{16} \,]$. It states that there is a number z (it will end up being the length of x minus 3) such that, if you divide the integer x by 16 ↑ z (thus effectively obtaining the first 3 hexadecimal digits of x) what you get is A0E, meaning that x starts with A0→ as needed.

H(x) is equally simple: $\exists z \, \exists w \, [x \text{ div } 16 \uparrow z = w*16 + F_{16} \,]$, that is, if you delete w symbols from the end of x then the last symbol of the result is F – that is, the freeze instruction F appears somewhere on the computation (presumably near the end), and thus M halts.

W(x) is a little more complicated. It states that for any three consecutive occurrences of E (that is, of →) in x, the two strings u and v between these occurrences represent configurations of M such that v is the next configuration after u:

$\forall z \, \forall z' > z \, \forall z'' > z' \, \forall w \, \forall w' \, \forall w'' \forall u \, \forall v \, [\, [x \text{ div } 16 \uparrow z = w*16 + E_{16} \text{ AND}$
$x' \text{ div } 16 \uparrow z = w'*16 + E_{16} \text{ AND } x'' \text{ div } 16 \uparrow z = w''*16 + E_{16} \text{ AND}$
$u = w \bmod 16 \uparrow (z' - z) \text{ AND } v = w' \bmod 16 \uparrow (z'' - z') \text{ AND } C(u) \text{ AND } C(v) \,]$
$N(u,v) \,]$

After the universal quantifiers of W(x), we have a sequence of conditions stating that there are →'s in the digits that are z, z', and z'' positions from the right end in the hexadecimal representation of x, and that u and v are the numbers whose hexadecimal representations are the strings between these →'s, and that u and v are configurations (this is what C(u) and C(v) mean), that is, they contain only one instruction symbol (A, B, C, D, or F) and no →'s. The precise details of formula C(u) are very simple and are left as an exercise. Then W(x) goes on to say that, if all those conditions are met, then v is the next configuration after u (this is what the formula N(u,v) means). The writing of N(u,v) is quite cumbersome, as it has to take into account the details of all instructions of M, and is also left as an exercise.