Constructing a Gödel Sentence

Stephen Lee[*]

September 22, 2011

Abstract

I argue that Gödel's incompleteness theorem is much easier to understand when thought of in terms of computers, and describe the writing of a computer program which generates the undecidable Gödel sentence.

1 Introduction

Gödel's incompleteness theorem is 80 years old, but is still seen as difficult to understand by many. In a time when computers are everywhere this should not be the case, as Gödel's theorem is very much about computers (Of course it is part of the genius of Gödel that he devised the theorem before electronic computers were invented). Gödel numbering is a way of storing a certain kind of data as bits and bytes. Gödel arithmetization should be familiar to anyone who knows how a compiler works. Then it's a case of following the diagonalization argument, and diagonalization should be familiar from other proofs (e.g. Cantor's proof of the uncountability of the real numbers)

Gödel's theorem states that given a first order axiomatization of arithmetic, such as the first-order Peano axioms, there exist arithmetical statements which cannot be either proved or disproved using those axioms. In this paper I describe a computer program which generates such an undecidable statement from the axioms. The program consists of a number of Java applets which can be found at [1]. There is a sketch of how the proof goes in this paper, but more details can be found in [1]. Much of this proof is based on the proof in [2], to which the reader who wants more details - or more rigor - is referred.

2 Encoding an arithmetic formula as a number

It is assumed that the reader is familiar with formal proofs from axioms, as explained in undergraduate level textbooks on mathematical logic, e.g.[3]. The important thing is that what counts as a statement is precisely defined, and that the rules of proofs are also precisely defined, so that checking a proof is a mechanical procedure.

In this project the encoding of an arithmetic statement into a number is done using the following tables. The expressions are built up using a Polish notation, so that the code for an operator comes first, followed by the codes for its parameters. Note however, that the bits of the number are read from right to left, i.e. the least significant bit first. Formulae and terms are encoded via the following tables

Formula	meaning	bits
$term = term$	equality	00
$\neg formula$	not	01
$formula1 \Rightarrow formula2$	implies	10
$\forall$**variable** $formula$	for all	11

Term	meaning	bits
variable	variable	0
0	zero constant	001
$term'$	successor function	011
$term + term$	addition	101
$term \times term$	multiplication	111

Variables can be x, y or z followed by a number. The letter is encoded as 00, 01 or 10 and then the following number is encoded in base-3 using the same bit pairs. The bit pair 11 is used to indicate the end of the number.

Thus the expression $0' + 0' = 0''$ is encoded as 100101101100101100101110100 in binary (which is 79059316 in decimal), since, reading from right to left gives:

[*]stephen@quantropy.org

00: term= term. This is followed by the code for the left hand term, which starts with 101:term+term. The left hand term has 011 (successor) followed by 001 (zero) and so is $0'$. This is followed by the right hand term of the addition, which is again $0'$. Finally comes the right hand term of the equality 001011011, which encodes $0''$. There is a 1 bit at the end to act as a terminator, although this is not strictly necessary.

The applet implementing this encoding can be found at Godel_applet1

3 Encoding proofs

Normally a proof consists of a series of mathematical statements, with the derivation of each from earlier ones given as a comment. When encoding a proof to be read by a computer, things are a bit different - it is the content of the comments which is the important part. Specifying how a statement is derived means that the computer can do the actual derivation. Hence each line of the proof is an axiom from the following table.

Axiom name	Axiom Structure	bits	Comment
Ax1(WF1,WF2)	$\mathtt{WF1} \Rightarrow (\mathtt{WF2} \Rightarrow \mathtt{WF1})$	0001	First are the three propositional axioms
Ax2(WF1,WF2,WF3)	$(\mathtt{WF1} \Rightarrow (\mathtt{WF2} \Rightarrow \mathtt{WF3})) \Rightarrow ((\mathtt{WF1} \Rightarrow \mathtt{WF2}) \Rightarrow (\mathtt{WF1} \Rightarrow \mathtt{WF3})$	0101	
Ax3(WF1,WF2)	$(\neg\mathtt{WF2} \Rightarrow \neg\mathtt{WF1}) \Rightarrow ((\neg\mathtt{WF2} \Rightarrow \mathtt{WF1}) \Rightarrow \mathtt{WF2})$	1001	
Ax4(V,T,WF)	$\forall\mathtt{V}\,\mathtt{WF(V)} \Rightarrow \mathtt{WF(T)}$ **(T is a term free for V in WF)**	0010	Next are two *predicate* axioms
Ax5(V,WF1,WF2)	$\forall\mathtt{V}\,(\mathtt{WF1} \Rightarrow \mathtt{WF2}) \Rightarrow (\mathtt{WF1} \Rightarrow \forall\mathtt{V}\,\mathtt{WF2})$ **(WF1 has no free occurence of V)**	0110	
Ind(V,WF)	$\mathtt{WF(0)} \Rightarrow ((\forall\mathtt{V}(\mathtt{WF(V)} \Rightarrow \mathtt{WF(V')})) \Rightarrow \mathtt{WF(V)})$	1010	Integer Induction
MP(L1,L2)	$\mathtt{WF1}, \mathtt{WF1} \Rightarrow \mathtt{WF2} \vdash \mathtt{WF2}$	0011	Finally two rules of inference
Gen(V,L)	$\mathtt{WF} \vdash \forall\mathtt{V}\,\mathtt{WF}$	0111	
End of proof		00	

The first six axioms take as parameters formulae, terms and variables which are encoded in a way similar to that specified in section 2. The final two axioms are slightly different. *MP* (Modus Ponens) takes two line numbers of earlier statements in the proof. These must be of the form `WF1, WF1` $\Rightarrow$ `WF2` and the derived statement is then `WF2` . *Gen,* which adds a universal quantifier to a statement, takes a line number and a variable. Instead of a line number these axioms can take an integer axiom from the following table.

No.	Axiom	Bits
I1	$x = y \Rightarrow (x = z \Rightarrow y = z)$	0001
I2	$x = y \Rightarrow x' = y'$	0011
I3	$\neg 0 = x'$	0101
I4	$x' = y' \Rightarrow x = y$	0111
I5	$x + 0 = x$	1001
I6	$x + y' = (x + y)'$	1011
I7	$x \times 0 = 0$	1101
I8	$x \times y' = (x \times y) + x$	1111

The program takes a proof encoded in this form and generates the statements of the proof, finishing with the statement to be proved. The applet is located at Godel_applet2

4 Gödel arithmetization

Having dealt with the straightforward matter of encoding statements and proofs in terms of bits, we now get on to a more central part of the proof, that is expressing a computer program in terms of arithmetic statements. In particular, given an integer function with one parameter f(x), written in a programming language, the task is to translate a statement such as y=f(x) into a formal arithmetical statement. The function to be translated in the proof of Gödel's theorem is that of Section 3, which takes the encoding of a proof as a parameter and returns the encoding of the statement proved as its result. This was written in Java, but I found that the conversion of program statements into arithmetic was much more straightforward if the functional programming language Haskell was used, so the proof decoding program was rewritten in this language (In fact a small subset of Haskell is used).

Formal arithmetic has no way of saying 'Do this n times', but such a concept is a crucial part of programming, for instance in expressing a function such as $f(n) = 2^n$, which would generally be written using a loop. In Haskell, however, loops are often implemented via recursion, so that this function can be programmed as

```
f n =if n==0 then 1 else 2*(f (n-1))
```

To convert this into an arithmetic statement requires another of Gödel's insights, the Gödel beta function which is defined as $\beta(a,b,x) = b\%(1+(x+1)a)$ where % is the modulo operator. It is straightforward to convert statements involving the β function into arithmetic statements. Now it is possible to show that given a sequence of natural numbers $a_1, a_2..a_n$, there exist numbers Y and Z such that $a_i = \beta(Y, Z, i)$ for all i in [0..n]. Hence it is possible to express $m = 2^n$ as follows

$$\exists Y, Z\ (m = \beta(Y,Z,n) \wedge (0 < i \leq m \Rightarrow \beta(Y,Z,i) = 2 * \beta(Y,Z,i-1)) \wedge \beta(Y,Z,0) = 1)$$

Further details of the arithmetization process can be found at Godel_arithmetization and the arithmetization applet can be found at Godel_applet3

5 The Gödel sentence

Given an arithmetical formula $\phi(x)$ with one free variable, and a numerical encoding *num* of aritmetical statements, it is possible to find a statement σ_ϕ such that σ_ϕ is equivalent to the statement $\phi(num(\sigma_\phi))$. This is known as Gödel's *fixed point theorem* and σ_ϕ is a *fixed point* of ϕ. (This is the diagonalisation step of the proof, although in the proof from[2] it doesn't appear to have much in common with Cantor's diagonal argument) From Section 3 we have a function `proof(x)` which takes an encoded arithmetical proof, and returns the statement which it proves. By Section 4 this can be expressed as an arithmetical statement **proof**. Now take $\phi(x)$ to be $\neg\exists y(x = *proof*(y)$. This has a fixed point σ_ϕ, which is then mathematically equivalent to

$$\neg\exists y(num(\sigma_\phi) = *proof*(y) \tag{1}$$

The claim is that σ_ϕ is the undecidable statement. For suppose that there existed a proof of σ_ϕ, then this proof could be encoded as an integer k. Then $num(\sigma_\phi) = *proof*(k)$, which, from 1 implies $\neg\sigma_\phi$, a contradiction. Hence there is no proof of σ_ϕ. But this is just what 1 is saying, and so σ_ϕ is true, meaning that there can be no proof of $\neg\sigma_\phi$.

The above assumes that the axioms are consistent, of course, but it also makes the stronger assumption that if $\psi(k)$ is false for all natural numbers k then there is no proof of $\exists y\ \psi(y)$, an assumption known as ω-consistency. This is true for the axioms given, but we might also want to consider other sets of axioms. The assumption can be dispensed with as follows. Define a function `proofsearch(x)`, which takes the encoding of an arithmetical statement μ, and searches through all proofs (which are ordered using the encoding used in Section 3) to see if there is a proof of μ, in which case it returns 1, or of $\neg\mu$, in which case it returns 0. Note that this function may not terminate, indeed we will find a case where it does not. Let $\phi'(x)$ represent `proofsearch(x)=0`, and let $\sigma_{\phi'}$ be its fixed point. Suppose that there were a proof of $\sigma_{\phi'}$. Then `proofsearch` would find it, so that $*proofsearch*(num(\sigma_{\phi'})) = 1$. However $\sigma_{\phi'}$ is mathematically equivalent to $*proofsearch*(num(\sigma_{\phi'})) = 0$, a contradiction. Alternatively if there were a proof of $\neg\sigma_{\phi'}$ then $proofsearch(num(\sigma_{\phi'})) = 0$, from which we could obtain a proof of $\sigma_{\phi'}$, a contradiction. Hence neither $\sigma_{\phi'}$ nor $\neg\sigma_{\phi'}$ have a proof from the given axioms.

Further details of this proof can be found at Godel_fixed_point.

6 Discussion

Putting all of the above together, it is possible to generate an undecidable sentence (as well as its Gödel encoding). These can be found at Godel_statement and Godel_number. Note that they are much shorter than might be expected. I had originally assumed that they might require tens or hundreds of megabytes, but the Gödel sentence has just 40203 characters and the Gödel number 29479 digits, so that they fit reasonably on a webpage. There have been other computer based proofs of Gödel's incompleteness theorem, e.g. [4], but I believe that this is the first one which has actually exhibited the undecidable statement which is generated.

There are a number of ways in which this project might be extended. Firstly, $\exists, \wedge$ and $\vee$ are not allowed in the arithmetic expressions - they need to be converted into expressions using just $\forall$ and $\Rightarrow$. It would be straightforward to allow a wider range of symbols. Secondly, the arithmetization program could be written in Haskell itself rather than Java (Haskell compilers are usually written in Haskell), and extended to allow a larger set of Haskell statements. The proof program has the integer axioms built into the program, but it could be modified so that the axioms could be given as input. It would then be possible to see the effect of adding the undecidable statement (or its negation) as an axiom to the program. Another possible extension of the proof program would be to make it easier to enter proofs - at present they have to be encoded by hand into their Gödel encodings. I feel that any of these extensions would be suitable for an undergraduate computer science project.

References

[1] S. Lee. Gödel's incompleteness theorem project. URL: http://tachyos.org/godel.html

[2] B. Kim. Complete proofs of gödel's incompleteness theorems. URL: http://web.yonsei.ac.kr/bkim/goedel.pdf

[3] H. B. Enderton, *A Mathematical Introduction to Logic, Second Edition.* Academic Press, 2001.

[4] N. Shankar, *Metamathematics, Machines and Gödel's Proof (Cambridge Tracts in Theoretical Computer Science).* Cambridge University Press, 1994.