Gödel's Incompleteness Theorem

Part I: Arithmetization

Computability and Logic

Gödel Numbering

- Let’s assign a natural number to FOL expressions such that we can recover the object from the encoding.
- This can be done in many different ways.

Gödel Numbering Scheme 1

Atomic variables ↓ (6th column) Constants ↓ (9th column)

( ) ,	$\neg$ $\wedge$ $\vee$ $\rightarrow$ $\leftrightarrow$	$\exists$ $\forall$ $\bot$	=	x_0, x_1, x_2, …	P^0_0 P^0_1 …	P^1_0 P^1_1 …	…	f^0_0 f^0_1 …	f^1_0 f^1_1 …	…
1 19 199	2 29 299 2999 29999	3 39 399	4	5 59 599 …	6 69 699 …	68 689 6899 …	…	7 79 799 …	78 789 7899 …	…

Just concatenate the numbers of each symbol to get number of any expression

E.g. PA3: $\forall$x x + 0 = x (which is really $\forall$x =(+(x,0),x)) has Gödel number 3954178815199719199519

Gödel Numbering Scheme 2

( ) ,	$\neg$ $\wedge$ $\vee$ $\rightarrow$ $\leftrightarrow$	$\exists$ $\forall$ $\perp$	=	x_i	P^n_i	f^n_i
1 3 5	7 9 11 13 15	17 19 21	23	$2*3^i$	$2^2*3^n*5^i$	$2^3*3^n*5^i$

E.g. the Gödel number for 0 is 8.

For expressions, use prime coding of sequence of numbers, whose entries are the Gödel numbers of the symbols of the expression.

E.g. the Gödel number for the expression 0 = 0 (which is =(0,0)) is

$2^6 * 3^{23} * 5^1 * 7^8 * 11^5 * 13^8 * 17^3$

Claim: The Set of all FOL formulas (and of all FOL sentences) is Recursive

- By this we mean that the set of Gödel numbers of all FOL formulas (sentences) is recursive.

Church-Turing Thesis

- The straightforward way to ‘prove’ the claim is by making an appeal to the Church-Turing Thesis.
- That is, it’s perfectly clear to us that there exists computer programs that can decide whether some string of symbols is a well-formed formula or sentence of FOL (assuming Fitch has no bugs, there is one!).
- So, by Church-Turing Thesis, there is a Turing-machine that can decide ‘wff-hood’ and ‘sentence-hood’, and hence the sets are indeed recursive.
- But we don’t need to make such a ‘lazy’ appeal.

Cryptographic Functions

- First, some functions dealing with the codings of strings of symbols.
- Where #s is the Gödel number of symbol string s:
- len(#s) = length of s
  - This is recursive, since len(x) = lo(x,2)
- ent(#s,i) = i-th entry of s
  - ent(s,i) = lo(s,π(i)) (remember, 3 is '1st' prime)
- last(#s) = last entry of s
- conc(#s,#r) = #(concatenation of s and r)
- ext(#s,a) = #(s with a added at the end)
- pre(#s,a) = #(s with a added at the beginning)
- sub(#s, c, d) = #(s with all entries c replaced by d)
- Proof that last 5 functions are recursive is left as HW.

Primitive Syntactical Properties

- Define LeftParen(x) = x is the code of a left parenthesis
- LeftParen(x) is recursive, since LeftParen(x) iff x = 1
- Similarly, the following sets are all recursive:
  - RightParen(x) iff x is code of a right parenthesis
  - Comma(x) iff x is the code of a comma
  - Neg(x) iff x is the code of a negation symbol
  - Conj(x) iff x is the code of a conjunction symbol
  - …

Primitive Syntactical Properties

- Predicate(x) = {x | x is the code number of a predicate symbol}
- Predicate(x) is recursive, since Predicate(x) iff $\exists i \Box x\ \exists n \Box x\ x = 2^2 * 3^n * 5^i$
- Similarly, the following sets are all recursive:
  - N-place-Predicate(x,n) iff x is code of n-place predicate
  - Variable(x) iff x is the code of a variable
  - Constant(x) iff x is the code of a constant
  - AtomicTerm(x) iff x is the code of an atomic term

Formation Sequences

- A complex term can be represented by a formation sequence: a sequence of expressions where each entry is either an atomic term or the application of a (n-place) function symbol to (n) earlier entries of the formation sequence.
  - E.g. s(0)*(s(0) * 0):
    - 1. 0 atomic
    - 2. s(0) s 1
    - 3. s(0) * 0 * 2,1
    - 4. s(0) * (s(0) * 0) * 2,3
- We can code this sequence of expressions the way we code other sequences: use prime encoding where each entry is the code of the expression belonging to that entry.
- The code of this sequence will be the code of the term

Term

- Term(x) iff x is the code of a term is recursive:
  - ComplexTerm(x) iff ∃n□x

    len(x) = n ∧

    ∀i < n ∃y□x y = ent(x,i+1) ∧

    ∃m□len(y)

    n-place-function(ent(y,1),m) ∧

    LeftParen(ent(y,2)) ∧

    RecursiveConc(x,y,m) ∧

    RightParen(last(y))
  - RecursiveConc(x,y,m) iff y is a concatenation of m symbol strings occuring in sequence x, separated by comma's iff

    (m = 1 ∧ ∃n□len(x) ent(x,n) = y) ∨

    (1 < m ∧ ∃v□y ∃w□y ∃n□len(x) ent(x,n) = v ∧ RecursiveConc(x,w,m-1) ∧ y = conc(z,pre(5,w)))
  - Term(x) iff AtomicTerm(x) ∨ ComplexTerm(x)

Atomic Formulas

- AtomicFormula(x) iff x is the code of an atomic formula is recursive as well (assuming no function symbols (i.e. not complex terms) and no identity symbol)
  - AtomicFormula(x) iff $\exists n$□ len(x)

    len(x) = 2*n + 2 $\wedge$

    N-place-Predicate(ent(x,1),n) $\wedge$

    LeftParen(ent(x,2)) $\wedge$

    $\forall$i < n AtomicTerm(ent(x,2*(i+1)+1)) $\wedge$

    $\forall$i < n-1 Comma(ent(x,2*(i+1)+2)) $\wedge$

    RightParen(last(x))