Metalogic Part II
Gödel's Remarkable Theorem

Presenter: Errol Martin

Metalogic

Part I of the metalogic course discussed (is discussing, will be discussing, ...)

- Axiomatisation and model theory of predicate logic
- Completeness Theorem for first-order predicate logic
- Formalisation of arithmetic as Peano Arithmetic in first order predicate logic.
- The Löwenheim-Skolem results about the size of first order models.

In Part II we cover the following topics

- Computability and Recursive Functions
- Proof that exactly the partial recursive functions are computable
- Gödel's Incompleteness Theorems

Lecture Topics

Historical Summary: Hilbert's Program; Gödel's Theorems; Formalisation of Arithmetic; Concept of Computability

Computable Functions I: Partial and Primitive Recursive Functions

Computable Functions II: Turing Machines

Church's Thesis

Theorem: Equivalence of partial recursive functions and Turing machine computable functions.

The incompleteness results: Arithmetisation of syntax

The incompleteness results: Main Theorems

References

These notes are mainly based on the texts of Boolos and Jeffrey, and Y.I. Manin. The article by Smorynski in the Handbook of Mathematical Logic also gives a good condensed coverage.

Boolos, G. and Jeffrey, R., *Computability and Logic*, Cambridge U.P. 1974 (third edition 1989).

Feferman, Solomon et.al.*Kurt Gödel: Collected Works*, Vol I, Oxford, 1986.

Manin, Y.I., *A course in mathematical logic*, Springer-Verlag, 1977.

Rogers, H., *Theory of recursive functions and effective computability*, McGraw-Hill, 1967.

Reid, C., *Hilbert*, Springer-Verlag, 1970. (Second Edition 1996)

Smith, Peter, *An introduction to Gödel's Theorems*, Cambridge U.P., 2007

Smorynski, C, *The incompleteness theorems*, in *Handbook of Mathematical Logic*, ed. Jon Barwise, North-Holland 1977.

A Brief History

- Hilbert's program;
- Gödel
- Computable functions: Church, Turing, Kleene

The very deep and very powerful results in metalogic of the 1930s were unexpected. They arose in a context in which it was expected that a finitary proof of consistency of arithmetic would shortly be forthcoming.

The Great Quest: Hilbert's Consistency Program

The mathematician David Hilbert (1862-1943) proposed the complete axiomatisation and formalisation of all mathematical knowledge and proofs.

Although committed to formal methods, many of Hilbert's proofs were existential in nature, which ran counter to the finitistic, constructivist methods of mathematics.

E.g., in 1886 David Hilbert had proved a conjecture in algebra called Gordan's Problem (Paul Gordan 18xx-1nubering). The proof was not satisfactory to all mathematicians, because it was non-constructive in its methods. It proved the existence of a basis for an algebra but did not show how to construct the basis.

Gordan responded: *"Das ist nicht mathematik. das ist theologie."*

Hilbert's Response

To deal with this criticism, Hilbert proposed that the formal methods program should establish that all of the Ideal existential arguments could in principle be replaced by Real constructive arguments, by showing some sort of conservation result:

Conservation Result

$$\mathbf{I} \vdash \phi \Rightarrow \mathbf{R} \vdash \phi$$

Consistency

Attempting to show that formal systems are consistent is a natural extension of the Conservation Program.

In the first place, consistency is the assertion that a certain string (e.g. $0 = 1$) is not derivable. Since this is finitistically meaningful it ought to have a finitistic proof.

More generally, proving consistency of the abstract, ideal, system, using finitistic means, already establishes the conservation result.

Consistency $\Rightarrow$ Conservation

Proof Idea:

Suppose $\mathbf{I}$ is some abstract theory and $\mathbf{R}$ is some real theory which proves the consistency of $\mathbf{I}$.

Thus the Conservation program reduces to the consistency program, and Hilbert asserted:

"If the arbitrarily given axioms do not contradict each other through their consequences, then they are true, [and] then the objects defined through the axioms exist. That, for me, is the criterion of truth and existence"

However, Gödel's results showed that this program does not work ...

Gödel's Incompleteness Theorem

The incompleteness theorems of Gödel (1931) undermined Hilbert's program. They depend on using arithmetic to code the metatheory of a formal theory into the formal theory itself. We discuss the details later. The first theorem, the Incompleteness Theorem, is:

Theorem. *Let* $\mathbf{T}$ *be a formal theory containing arithmetic. Then there is a sentence* φ *which (under coding) asserts its own unprovability and is such that*

(i) $\mathbf{T}$ *is consistent* $\Rightarrow$ $not(\mathbf{T} \vdash \varphi)$.

(ii) $\mathbf{T}$ ω*-consistent* $\Rightarrow$ $not(\mathbf{T} \vdash \neg\varphi)$

Intuitively, the sentence φ is true, since, assuming that $\mathbf{T}$ is consistent, it is unprovable and it 'says' that it is unprovable. However, it is not a theorem of $\mathbf{T}$, assuming that $\mathbf{T}$ is consistent. Hence $\mathbf{T}$ is incomplete on this (practically necessary) assumption.

For discussion: Is it reasonable to assume that $\mathbf{T}$, viz. formalised arithmetic, is consistent?

Gödel's Second Incompleteness Theorem

Theorem. *Let* $\mathbf{T}$ *be a consistent formal theory containing arithmetic. Then*

$$not(\mathbf{T} \vdash \mathbf{ConT})$$

where $\mathbf{ConT}$ *is the (coded) sentence asserting the consistency of* $\mathbf{T}$*.*

This theorem directly affects the consistency program.

Formalisation of Arithmetic

Peano had proposed axioms for arithmetic in the 19th century. It turns out that these can be given a first-order formalisation.

Peano Arithmetic PA:

Take a first order predicate language with one individual constant 0 (read: zero) and one unary function $s(x)$ (read: the successor of x). The numbers are coded by 0, $s(0)$, $s(s(0))$, etc.

Peano Arithmetic is an extension of first-order logic which adds to the axiomatisation of logic additional axioms defining the properties of numbers. This can be done using the language of first-order logic.

The concept of *function computable by an algorithm*

Around 1935 the informal notion of an algorithmically computable function was formalised in several ways, including simple step-at-a-time calculations (Turing Machines), and building up (recursively defining) functions starting from a very simple basis.

Informally a function $y = f(x_1, \ldots, x_k)$ is computable if there exists a procedure or algorithm which determines its value in a finite number of steps.

Because we are formalising an informally given concept, there is always the possibility of another definition of computability, and the possibility that it might not be equivalent to the previously established theories of computability.

However, it turned out that all of the formal proposals for computability are equivalent: they pick out the same set of functions.

This became the subject of much discussion and analysis in the years following the publication of Gödel's results, with proposals by Kleene, Markov, Church, and others ...

Attributes of Computable Functions

Hartley Rogers (*Theory of Recursive Functions and Effective Computability*) lists 10 features which are relevant in analyzing the informal notion of an algorithm:

1. Finite set of instructions
2. A computing agent carries out the instructions
3. The steps can be stored and retrieved
4. The agent carries out the instructions in a discrete stepwise manner' (i.e. no fuzzy logic!)
5. The agent carries out the instructions deterministically
6. No fixed bound on the size of the inputs
7. No fixed bound on the size of the instruction set
8. No fixed bound on the size of working storage
9. The capacity of the computing agent is to be limited, normally to simple clerical operations
10. There is no fixed bound on the length of the computation.

Of these, only # 10 is contentious. According to Rogers, some mathematicians find counterintuitive certain theorems in the formal theory of computability which embody # 10.

We will examine three formalisations of computable functions:

- Partial Recursive Functions – An ‘axiomatic approach’
- Turing Machine (computable) Functions – A ‘state-machine’ approach
- Abacus Machines – A ‘computer-like’ approach

and discuss and outline the proofs that they are equivalent

Church's Thesis

Alonzo Church proposed the thesis that the set of functions computable in the sense of Turing Machines or partial recursive functions is identical with the set of functions that are computable by whatever effective method, assuming no limitations on time, speed, or materials.

Church's Thesis (p.20, Boolos and Jeffrey)

"But since there is no end to the possible variations in detailed characterizations of the notions of computability and effectiveness, one must finally accept or reject the *thesis* (which does not admit of mathematical proof):

Thesis: the set of functions computable in one sense is identical with the set of functions that men or machines would be able to compute by whatever effective method, if limitations on the speed and material were overcome."

Recursive Functions

Recursive functions are a sort of 'axiomatic' development of the concept of computability. We will follow Rogers' approach to recursive functions:

- Define the primitive recursive functions first.
- Then show that the primitive recursive functions are insufficient to be all of the algorithmically computability functions, because of diagonalisation and the existence of strong counterexamples.
- Introduce the partial recursive functions as a remedy for this.

Three approaches to Computability

- Primitive and Partial Recursive Functions
- Turing Machines
- Abacus (Register) Machines
- Diagonalisation, Halting, Undecidability

Primitive Recursive Functions

A function $f(\overline{x}) = f(x_1, \ldots, f(x_n)$ on the integers is *primitive recursive* iff it is either a *basic* function:

$$f(x) = 0$$
$$f(x) = s(x)$$
$$f(\overline{x}) = x_i$$

or the function is defined from other primitive recursive functions by the *rules* of *composition*:

$$f(\overline{x}) = g(h_1(\overline{x}), ..., h_m(\overline{x})$$

and *recursion*:

$$\begin{cases} f(0, \overline{x}) & = g(\overline{x}) \\ f(x+1, \overline{x}) & = h(f(x, \overline{x}), x, \overline{x}) \end{cases}$$

What functions are primitive recursive?

- All constant functions

  $f(x) = 3 =_{df} s(s(s(0)))$ is defined by iterating successor and composition

  $f(x) =_{df} s(s(s(0)))$

- Addition, Multiplication, Exponentiation

  Using the recursion rule

- Functions defined by cases:

$$f(\overline{x}) = \begin{cases} g_1(\overline{x}) \text{ if } h(\overline{x}) = 0 \\ g_2(\overline{x}) \text{ if } h(\overline{x}) \neq 0 \end{cases}$$

- ...

- Continuing in this way, developing the argument that various functions are *primitive recursive*, it becomes plausible that all ordinary mathematical functions are primitive recursive.

- ... However, this is not quite true

Partial Recursive Functions

The class of primitive recursive functions is *insufficient* to define all computable functions, since it lacks the case where a function is implicitly defined within another function.

The partial recursive functions add another rule:

$$h(\overline{x}) = \begin{cases} \text{smallest } y \text{ s.t. } f(\overline{x}, y) = 0, \text{if it exists} \\ \text{undefined, otherwise} \end{cases}$$

Notation: $h = Mn[f]$

Example: $Mn[sum]$

$$Mn[sum](x) = \begin{cases} 0 \text{ when } x = 0; \\ \text{undefined otherwise} \end{cases}$$

Turing Machine Computability

Provides a 'state-machine' flavour to computation. A Turing Machine is well-known. Consists of:

- An unending tape marked into squares
- Symbols: $s_0 =$ blank, $s_1, ..., s_n$
- A state-machine that, depending on its current state, reads/writes the current square and possibly changes to a new state
- States q_0,...,q_m
- Actions:

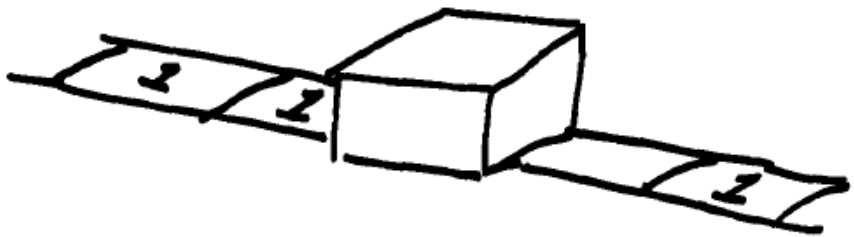
  Halt

  Move Right one square

  Move Left one square

  Write s_i on the current square ($o \leq i \leq m$

‘Snapshot’ of a Turing Machine



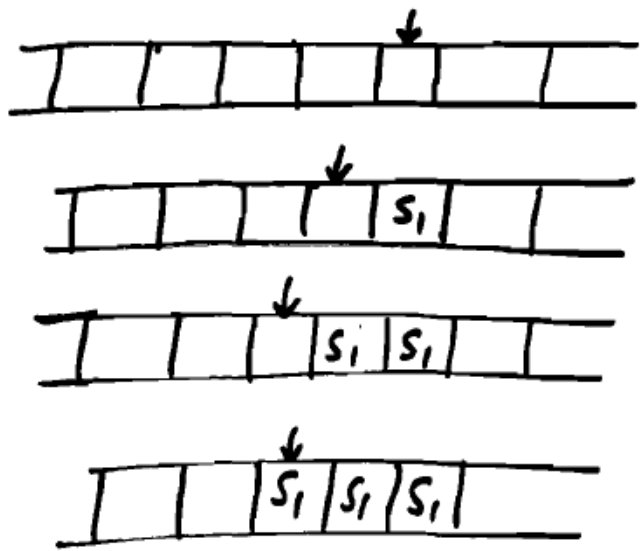
At most a finite number of squares are not blank, both initially and at later stages

The contents of the current (scanned) square is is known to the machine

The initial tape configuration forms part of the description

Machine starts in state $q1$, by convention

Example: Write $s_1 s_1 s_1$ onto a blank tape

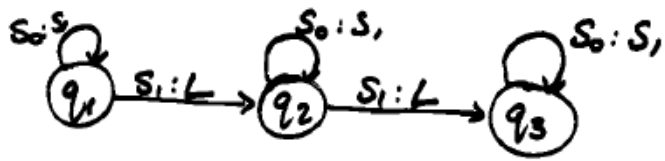


Specifying a Turing Machine program

1. *State-Event table*. What the machine does for each possible state and scanned symbol:

	s_0	s_1
q_1	$s_1 q_1$	$L q_2$
q_2	$s_1 q_2$	$L q_3$
q_3	$s_1 q_3$	

2. *Flow graph*



3. To be really official about this, define a Turing Machine TM as a *Set of quadruples*:

$TM =<$ present state, scanned symbol, action, next state $>$

In our case, TM_3, the TM which writes 3 1s on a blank tape then stops, is:

$$TM_3 = \{ < q_1, s_0, s_1, q_1 >, < q_1, s_1, L, q_2 >, < q_2, s_0, s_1, q_2 >, \\ < q_2, s_1, L, q_3 >, < q_3, s_0, s_1, q_3 > \}$$

Ways to Represent Turing Machines

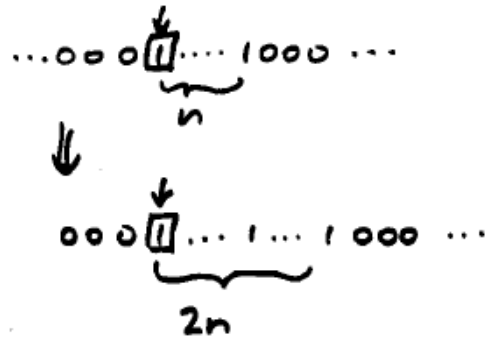
1. Set of quadruples as above

2. Redefine q_1 as q', q_2 as q'', s_5 as s'''', etc. Then the set of quadruples defining a Turing machine can be represented as a single 'word' constructed on a finite alphabet according to specific construction rules: need some standard convention to describe halting.

$$q'SS'q'q'S'Lq''$$

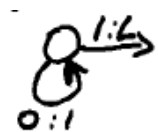
3. Sequence of Configurations: The computation can be represented by keeping track of the tape configurations.

Turing Machine Example: Double the number of 1's on a tape

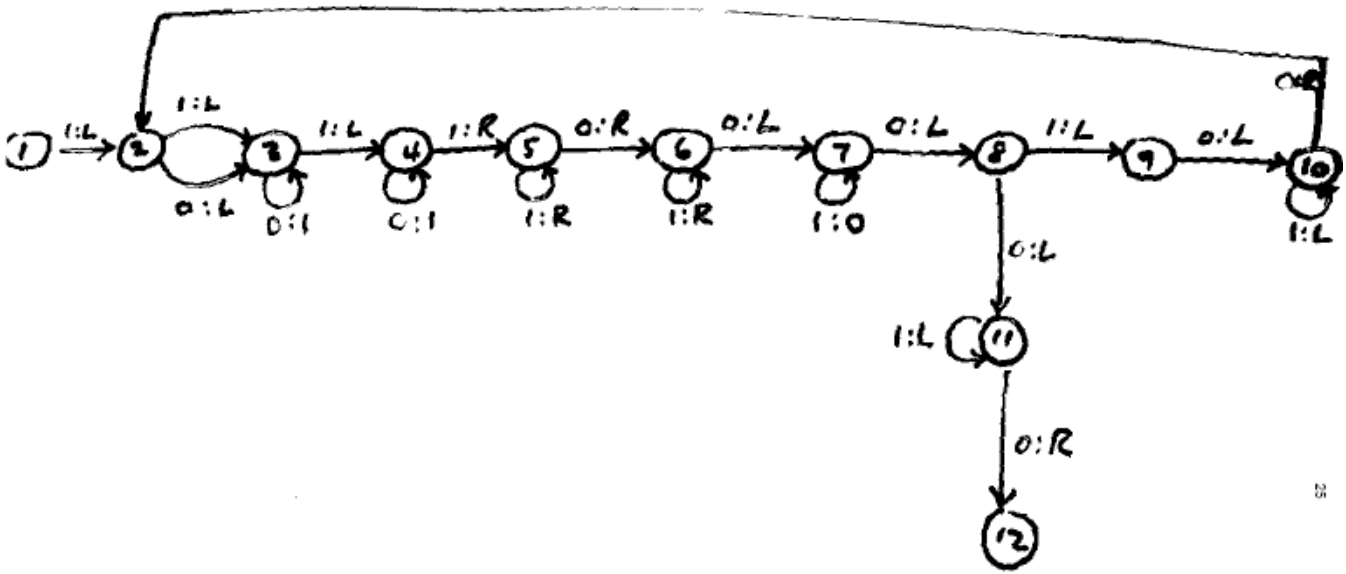


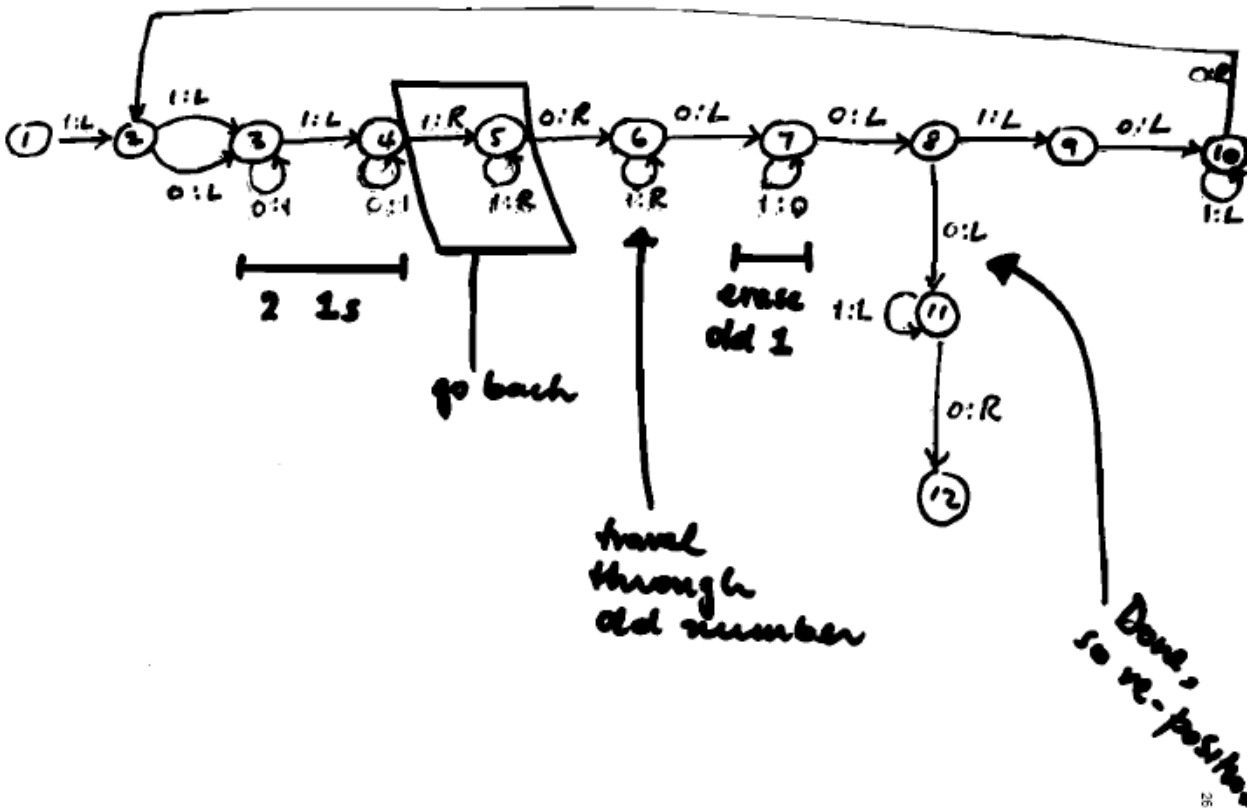
TM Example: Write 2n 1s on an initially blank tape

Method 1: String together 2n replicas of



Method 2: Write n 1s on the tape, then double that (using the previous program)


1
2
3
4
5
6
7
8
9
10
11
12
1:L
1:L
0:L
0:1
1:1
1:R
0:R
0:L
0:L
1:L
0:L
1:L
1:R
1:R
1:0
0:L
1:L
0:R


1
1:L
2
1:L
0:L
3
1:L
0:1
4
1:L
1:R
5
0:R
1:R
6
0:L
1:R
7
0:L
1:0
8
1:L
9
0:L
10
1:L
0:L
11
1:L
0:R
12
2 1s
go back
erase old 1
travel through old number
Done, so re-position

These programs quickly become immensely complicated; fortunately we will shortly move to arguments about TMs, rather than using them for computation.

Diagonalisation

This well-known method shows that for any enumeration $S_1, S_2, ...$ of sets (of integers, say), a set can be constructed which is not on the list.

Diagonalisation Method

Use a characteristic function s_i to represent the set S_i:

$$s_i(n) = \begin{cases} 1 \text{ if } n \in s_i \\ 0 \text{ otherwise} \end{cases}$$

The enumeration $S_1, S_2, ...$ is then

$$\begin{array}{lll} 1.s_1(1) & s_1(2)s_1(3) & ... \\ \vdots & & \\ m.s_m(1) & s_m(2)s_m(3) & ... \\ \vdots & & \end{array}$$

Now construct the function $S(n)$:

$$S(n) = \begin{cases} 0 \text{ if } s_n(n) = 1 \\ 1 \text{ if } s_n(n) = 0 \end{cases}$$

That is, $S(n)$ takes the *opposite* of the (n, n) diagonal value. Result: S is not in the enumeration

The Halting Problem

An application of diagonalisation.

Any Turing Machine M (a set of special quadruples, as above) can be represented as a 'word' in an alphabet of 6 letters: $\Box$, 1, $'$, q, R, L

For example, the machine $\{q_0 1 R q_1, q_1 1'' 1' q_2\}$ maps to $q1Rq'q'1''1'q''$

This can be coded into the standard language

$$\begin{aligned} \Box &\Rightarrow s_0 \\ 1 &\Rightarrow s_1 \\ ' &\Rightarrow s_2 \\ q &\Rightarrow s_3 \\ R &\Rightarrow s_4 \\ L &\Rightarrow s_5 \end{aligned}$$

Using this coding, every TM M has a *standard description* $\ulcorner M \urcorner$

The Halting Problem

Question:

Does machine M eventually stop when given the input $\ulcorner M \urcorner$? (I.e. when started on a tape with $\ulcorner M \urcorner$ written on it?)

Suppose a machine S existed which:

- Takes $\ulcorner M \urcorner$ as input, and
- Eventually stops on a 1 if M does halt when given $\ulcorner M \urcorner$, and
- Stops on a blank if M never stops when started with input $\ulcorner M \urcorner$.

Question:

Does machine S stop on $\ulcorner S \urcorner$

$\Rightarrow$ leads to a contradiction

Universal Turing Machine

It is possible to construct a machine U which will simulate the action of any machine M.

U takes as input:

- A standard description $\ulcorner M \urcorner$
- A coding of a tape pattern

The question

Does U applied to a word W eventually stop on a $\Box$?

is *unsolvable*

Undecidability

The unsolvability of the general halting problem can be applied to the problem of the decidability of a logical theory:

Code U to a formula φ

The effect of U on given inputs is expressed as logical consequences of φ.

Hence: If (sufficiently rich) logical theory $\mathbf{T}$ is decidable, then the halting problem for U is solvable.

Thus: Logical Theory is *undecidable*

Lambek's Abacus Machine

This is our third approach to computability. Will allow us to connect unsolvability/undecidability to incompleteness.

A Lambek machine is a register style machine similar to a digital computer in formal description.

Consists of an unlimited number of registers

containing numbers of arbitrary size.

There are two primitive operations:

- Add 1 to a register
- Remove 1 from a register (or emit an exception if the register contains 0

Primitive Operations are connected together and this is represented with a flowchart. The flowcharts for the two primitive operations are:

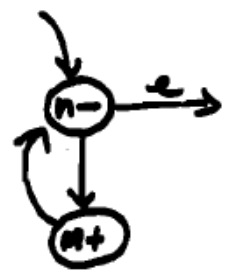
Add 1 to Register R_n:

Remove 1 from Register R_n (or emit an exception).

Addition Using an Abacus Machine

A flow chart to represent:

Add the numbers in R_n and R_m and leave the result in R_m would look like this:



Exercise: Add the numbers in R_n and R_m without loss from R_n

Equivalence of the various forms of computability

Theorem. *A function $f(\overline{x})$ is partial recursive iff $f(\overline{x})$ is Turing Machine computable*

Proof (Boolos and Jeffrey, Kleene, Wang, Lambek). Use 3 *lemmas*

$$\mathbf{A} \subseteq \mathbf{T}$$
$$\mathbf{R} \subseteq \mathbf{A}$$
$$\mathbf{T} \subseteq \mathbf{R}$$

where $\mathbf{A}$ is the set of functions computable by Abacus Machines, $\mathbf{T}$ is the set of functions computable by Turing Machines, and $\mathbf{R}$ is the set of partial recursive functions.

[Preliminary step] Set up a canonical form for TM computations.

Suppose a TM is computing $f(x_1, x_2)$ for given arguments. The computation can be arranged as follows:

1. The arguments x_1, x_2 are represented by blocks of 1s separated by a single blank. The value $f(x_1, x_2)$ is also a single block of 1s.

2. Machine starts and stops at leftmost 1.

3. Machine never moves more than two squares beyond the leftmost 1.

4. Machine writes only 1 and $\Box$ (blank).

5. The value (result of the calculation) is written starting from the same square as the leftmost 1 of the arguments.

This form is arbitrary, but fixes the computation for translation into other forms of computability.

In a similar way, we need to specify a standard configuration for an abacus machine. When calculating $f(x_1, \ldots, x_n) = y$:

Put $f(x_1, \ldots, x_n)$ into the first n registers (or, n adjacent registers if the first doesn't make sense). The remaining registers are to be empty. The result y is found in some specified register R_m, $m \neq 1, \ldots, n$

Lemma. $\mathbf{A} \subseteq \mathbf{T}$

Proof Method. The idea is to translate the initial configuration of the abacus registers to a standard Turing machine tape.

Describe a method to translate the *flowgraph* of the abacus machine to *Turing Machine quadruples*.

The main cases are for the basic operations of 'add to a register' and 'take from a register':

The ‘add to a register’ case

translates to:

- From standard position, move to the blank at the end of block s
- Write a 1
- Move right to check if more blocks
- If s is the last block return to standard position
- If more blocks then move move them all one square to the right, and
- Return to standard position.

The 'remove 1 from register if you can' case

translates to:

- From standard position, move to the first 1 in the s-th block
- If this is a single 1 (test by moving right one space) then $[S] = 0$.
- If $[s] = 0$ return to standard position
- If $[s] \neq 0$ erase the rightmost 1
- Move the remaining blocks, if any, one square to the left
- Return to standard position.

Mop-Up Operation

Translating the abacus operations leaves the value $f(x_1, \ldots, x_n)$ in the nth block.

The n-th must be moved to standard position:

- Leave a marker at the standard position
- Erase all other 1s except for the n-th block
- Move the n-th block up to the standard position

Lemma. *The partial recursive functions are abacus computable*

$$\mathbf{R} \subseteq \mathbf{A}$$

Proof. Proof method

Find flow graphs for each of the basic functions

Show how to convert flow graphs corresponding to the arguments of the two rules o composition, recursion and minimisation, into a flow graph computing the result.

– composition

– recursion

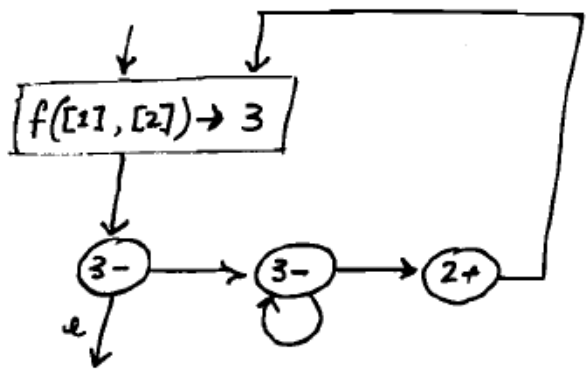
– minimisation

Minimisation

Compute $M_n[f](x)$, given $f(x, y)$

Put x in register 1

Put 0 in register 2



Compute $f(x, 0)$ if $= 0$ then done

else

Compute $f(x, 1)$ if $= 0$ then done

else

...

Theorem (Kleene). *In obtaining a recursive function from basic functions using the composition, recursion and minimisation rules, the operation of minimisation need not be used more than once.*

Completing the circle of equivalences

Lemma. *All functions computable by means of a Turing Machine are partial recursive*

$$\mathbf{T} \subseteq \mathbf{R}$$

Proof. Consider a Turing Machine implementing a standard computation

Associate a *left number* and a *right number* with each stage of the computation

Show that the left and right numbers are *primitive recursive functions* of the initial left and right numbers

Use minimisation to model the situation that the machine halts (enters state 0, say) after computing f

Proof that $\mathbf{T} \subseteq \mathbf{R}$ continued

Now define $g(a, b, c)$

If t is a stage not later than the stage at which M halts when computing $f(x_1, x_2)$

$$g(x_1, x_2, t) = \langle \text{left no., state, right no.} \rangle$$

At beginning,

$$g(x_1, x_2, 0) = \langle 0, 1, s(x_1, x_2) \rangle$$

[where 0 is the left number, 1 is the state, and s(x1,x2) is the right number.]

By definition

$$ctr(g) \neq 0 \text{ for all } y \leq t$$

Thus M halts at t when computing $f(x_1, x_2)$ iff t is the least stage y s.t. $ctr(g) = 0$.

Use M_n to represent this.

Gödel Numbering

Now, what does all this have to do with logic, arithmetic, and incompleteness? Start by connecting up arithmetic terms (which are modelled as numbers) with formulas in logical symbolism.

Coding Method which assigns a closed term $t = \ulcorner \varphi \urcorner$ to each formula φ in the language.

Also can be extended to assign a closed term to each *derivation* – which after all is only a sequence of formulas.

If the language in question is the language of formalised arithmetic, then the code will be a numeral

$$n =_{\text{df}} \underbrace{s(s(\cdots s(0)\cdots)}_{n \text{ successor symbols}}$$

Coding Details

The coding method has to assign a unique term to each formula. There are many ways to do this. Here is the approach due to Boolos and Jeffrey.

(	)	&	$\exists$	x_0	f_0^0	f_0^1	f_0^2	$\cdots$	A_0^0	A_0^1	A_0^2
1	2	3	4	5	6	68	688		7	78	788
	,	$\wedge$	$\forall$	x_1	f_1^0	f_1^1					
	29	39	49	59	69	689					
					f_2^0	f_2^1					
					699	6899					

Example: $\forall x(x = x)$ is coded by this scheme to 4951578852

Proof is required that this representation is unique.

The classical coding mechanism

Gödel used a coding method based on unique composition into primes.

Symbol	Code
0	1
s	2
$+$	3
$\times$	4
$=$	5
(	6
)	7
x	8
$'$	9
$\neg$	10
$\wedge$	11
$\exists$	12

Example: $\neg(\exists x)\neg(x = x)$ is coded as $2^{10}.3^{6}.5^{12}.7^{8}.11^{7}.13^{10}.\cdots p_n^{()}$.

This representation is unique, by the fundamental theorem of arithmetic.

Official gödel numbering for derivations

Note that the gödel number of a formula φ, viz $\ulcorner\varphi\urcorner$, must be greater than 12. This lets us continue the same general scheme for derivations:

Sequences of formulas

$$\varphi_1, \varphi_2, \ldots$$

can be coded by

$$2^{gn(\varphi_1)}.3^{gn(\varphi_2)}.\ldots.p_n^{gn(\varphi_n)}$$

Hence, derivations can be coded as well as formulas

Representing Gödel Numbers in Arithmetic

Let $\ulcorner\varphi\urcorner$ be the *numeral* corresponding to the gödel number of φ. ($\ulcorner\varphi x\urcorner$ is a closed term).

A Turing Machine (hence a recursive function by previous theorem) can calculate whether n is the gödel number of an expression

Recursive functions can be defined formally (i.e. in formal arithmetic) leading to

A function

$$sub(\ulcorner\varphi x\urcorner, t) = \ulcorner\varphi t\urcorner$$

A provability predicate

$$\mathsf{Prov}(t_1, t_2)$$

$$\vdash \varphi \text{ iff } \vdash \mathsf{Prov}(t, \ulcorner\varphi\urcorner)$$

for some closed t

Provability Predicate

$\vdash_{\mathbf{T}} \mathrm{Prov}(t_1, t_2)$ means, informally, "t_1 is the code of a derivation in $\mathbf{T}$ of the formula with code t_2".

Suppose we further define:

$$Pr_T(y) =_{df} \exists x Prov_T(x, y)$$

Now we have reached a truth-predicate for a formula, a model. Therefore, can test the adequacy of the model by asking the question:

Is $\mathbf{T} \vdash \varphi \iff \mathbf{T} \vdash \mathbf{Pr}(\ulcorner \varphi \urcorner)$?

From left to right:

obtained from

$$\mathbf{T} \vdash \varphi \iff \mathbf{T} \vdash \mathbf{Prov}(\mathbf{t}, \ulcorner \varphi \urcorner) \text{ for some } \mathbf{t}$$

by generalisation

From right to left. This is more difficult. It may not be true, unless there is a *numeral* for each number which is a code (in the right way).

Derivability Conditions

The encoding program can be carried out in the formal system so that:

D1. $\mathbf{T} \vdash \phi \Rightarrow \mathbf{T} \vdash \mathbf{Pr}(\ulcorner \phi \urcorner)$

D2. $\mathbf{T} \vdash \mathbf{Pr}(\ulcorner \phi \urcorner) \rightarrow \mathbf{Pr}(\ulcorner \mathbf{Pr}(\ulcorner \phi \urcorner) \urcorner)$

D3. $\mathbf{T} \vdash (\mathbf{Pr}(\ulcorner \phi \urcorner) \wedge \mathbf{Pr}(\ulcorner \phi \rightarrow \psi \urcorner)) \rightarrow \mathbf{Pr}(\ulcorner \psi \urcorner)$

Diagonalisation Lemma

Let $A(x)$ be a formula with only x free. Then there is a formula G such that

$$\vdash G \leftrightarrow A(\ulcorner G \urcorner$$

Proof.

Let $Bx \leftrightarrow A(sub(x, x))$ be the *diagonalisation* of A.

Let $m = \ulcorner Bx \urcorner$

Let $G = Bm$

Then

G	$\leftrightarrow Bm$	Df.
	$\leftrightarrow A(sub(m, m))$	Df.
	$\leftrightarrow A(sub(\ulcorner Bx \urcorner, m))$	Df m.
	$\leftrightarrow A(\ulcorner Bm \urcorner)$	Df sub.
	$\leftrightarrow A(\ulcorner G \urcorner)$	Df.

Now apply the diagonalisation lemma to the formula $\neg Pr(x)$

Theorem. *Suppose A is a sentence which "asserts its own unprovability", i.e. $\vdash A \leftrightarrow \neg Pr(\ulcorner A \urcorner)$*

then

(i) $\mathbf{T} \not\vdash \mathbf{A}$

(ii) $\mathbf{T} \not\vdash \neg\mathbf{A}$ *provided* $\mathbf{T}$ *is ω-consistent*

Proof of (i).

$$\begin{aligned} \mathbf{T} \vdash A &\Rightarrow \mathbf{T} \vdash \mathbf{Pr}(\ulcorner \mathbf{A} \urcorner) && (D1) \\ &\Rightarrow \mathbf{T} \vdash \neg\mathbf{A} && (hyp) \\ &\Rightarrow \mathbf{T} \text{ is inconsistent} \end{aligned}$$

thus $\mathbf{T} \not\vdash \mathbf{A}$

Proof of (ii)

If T is ω-consistent, then the converse of D1 holds, viz:

$$T \vdash Pr(\ulcorner A \urcorner) \Rightarrow T \vdash A$$

$T \vdash \neg A$ is proved in (i)

$\Rightarrow T \vdash \neg\neg Pr(\ulcorner A \urcorner)$

$\Rightarrow T \vdash Pr(\ulcorner A \urcorner)$

$\Rightarrow T \vdash A$

$\Rightarrow$ contradiction