TECHNICAL PAPER

A Simple Generator of Incompleteness Theorems (Written in M of Course)

by Charlie Volkstorf

Abstract

We present Turing's proof of the unsolvability of the Halting Problem in the context of M routines. A closer examination of Turing's proof reveals a set of nine program transformations that can be applied to any program, to create a program that is sometimes inherently inconsistent. This process is generalized, formalized and implemented in a 42-line M routine. Running this routine produces lists of problems that the routine discovers are solvable or unsolvable. This includes the Halting Problem and several other well-known theorems traditionally proven using informal, manual methods.

I. Paradoxes, Self-Reference, Incompleteness

There is no topic in mathematics (computer science) that is more intriguing than the paradoxes that have plagued man since antiquity. These are proofs that certain statements, that seem to be false, are in fact true. While many of them merely illustrate flaws in the statement of the paradox, others have uncovered fundamental limitations (incompleteness) in various branches of mathematics and have spawned significant areas of investigation. The most notable include:

1. Russell's Paradox: Philosopher Bertrand Russell asked in 1901 [5], "Does the set of all sets that do not contain themselves contain itself ?" By its definition, this statement is neither true nor false. (Neither the statement nor its negation is true.)

2. Godel's 1st Incompleteness Theorem: Mathematician Kurt Godel proved in 1931 [5] that typical systems of logic contain statements that are true but not provable. (Neither the statement nor its negation can be proven.)

3. Unsolvability of the Halting Problem: In 1937, mathematician Alan Turing [1] defined a simple, but completely general model of computer programming, later called a Turing Machine. He proved that it is impossible for any Turing Machine to determine whether or not an arbitrary Turing machine, with its input, will ever halt.

II. The Halting Problem

The unproven premise of Turing Machines is that any computation that can be done can be done with a Turing Machine. Thus any requirement that can't be carried out by a Turing Machine (such as the Halting Problem) can't be done at all. We will assume the equivalence of Turing Machines and M extrinsic functions with parameter passing, with the following correspondence:

1. The Turing Machine tape (input) is represented by the formal parameters passed to the M extrinsic function.

2. A Turing Machine halting YES or NO is represented by the M extrinsic function returning 1 or 0 (using QUIT with an argument), respectively.

3. As we are concerned with the computational power of M and not how information is transferred between users and M, we consider the subset of M that excludes the READ and WRITE commands.

Let us represent a program within M as a single string (as is done in some M implementations). Now suppose that we have a solution to the Halting Problem, that is, an extrinsic function defined as:

```
H(P1,P2)   ; Determine whether program
P1 halts when P2 is passed to it as a
parameter.
       . . .  QUIT 1 or QUIT 0
```

Now consider the following one-line subroutine in the same routine:

```
H2(A)     N (A) Q:'$$H(A,A) 1 F
```

What is it doing ? It takes an input A and merely returns 1 if `'$$H(A,A)`, and loops otherwise. Thus `H2(A)` halts if A loops on itself, and `H2(A)` loops if A halts on itself. `H2` does the opposite of what its input, interpreted as being the representation of an M extrinsic function, does on itself, with respect to halting or not.

Now consider a third subroutine in this routine:

```
H3 Q $$H2($T(H2))
```

All this does is to call H2 with input that happens to be the representation of subroutine H2. What would this do? As established above, H2(A) halts if A loops on itself and loops if A halts on itself. Since the input A to H2 is H2 itself, then H2 halts on itself if H2 loops on itself, and loops if it halts on itself. Both of these are inconsistent, so H2 can neither halt nor loop. This is impossible, so these programs cannot exist. Since H2 and H3 do exist, H cannot exist. Thus there is no solution to the Halting Problem.

Note that H2 and H3 could be combined to produce one inconsistent subroutine, H4:

```
H4  Q:'$$H("H2(A) N (A) Q:'$$H(A,A) 1 F"
, "H2(A) N (A) Q:'$$H(A,A) 1 F") 1 F
```

III. Formalization of the Properties of Turing Machines

The above is more or less standard Theory of Computation talk. The purpose of this note is to utilize M to extend these results. In particular, we formalize these concepts to the point of being able to write an M routine to carry out the synthesis of these, and other theorems.

For any predicates P, Q, R, we say that a Turing Machine solves requirement P:Q:R if it halts YES if and only if P holds, halts NO if and only if Q holds, and loops if and only if R holds. (Thus P, Q, R must be exclusive and exhaustive.) Furthermore, +P:Q:R means that there is a Turing Machine that solves P:Q:R, and -P:Q:R means that no such Turing Machine exists.

We define the following relations, using an English statement expressing when each holds:

YES(**a,b**)—Turing Machine **a** eventually halts YES when started on input **b**.
NO(**a,b**)—Turing Machine **a** eventually halts NO when started on input **b** .
HALT(**a,b**)—Turing Machine **a** eventually halts when started on input **b** .
LOOP(**a,b**)—Turing Machine **a** when started on input **b** never halts.
TRUE(**a**)—**a** is any value.

We use symbols 0 and 1 to represent false and true, respectively.

Finally, following the notation introduced in [6], we define, for any predicate P, expressions +P(I), +P(x),-P(I) and -P(x) as follows. (Expressions of the form +P(I,x) or -P(I,x) are not needed in the present note.)

Expression +P(I)means that we have a program that decides if P holds for any given value. Thus, there is a Turing Machine that, when started on an input of I, halts YES if P(I), and halts NO if ~P(I). (P is recursive.)

Expression +P(x) means that we have a program that lists out the values x for which P(x) holds, (i.e., { x | P(x) }). (P is recursively enumerable.)

The expressions -P(I) and -P(x) are the formal negations of P(I) and P(x), respectively.

+P(I) is obviously equivalent to +P(I):~P(I):0. It is well known [2] that +P(x) (a set being recursively enumerable) is equivalent to there being a Turing Machine that halts YES on elements of the set and loops otherwise. Thus +P(x)is equivalent to +P(I):0:~P(I).

IV. Formalization of Turing's Theorem

Turing's Theorem, the unsolvability of the Halting Problem, is -HALT(I,J). But how do we formally prove that -HALT(I,J)?

By definition, H solves HALT(I,J):~HALT(I,J):0 establishing +HALT(I,J). By construction, H2(I) returns 1 when ~HALT(I,I) and loops otherwise. Thus, H2 solves ~HALT(I,I):0:HALT(I,I) establishing +~HALT(x,x).

Turing proved -HALT(I,J) by constructing H2 and H3 from H, then showing an inconsistency. The transformation from H to H2 may be written as:
HALT(I,J):~HALT(I,J):0 =>
~HALT(I,I):0:HALT(I,I).

There are three parts to this transformation: (1) replace I,J with I,I (2) direct a return value of 1 from H to a LOOP in H2, (3) direct a return value of 0 from H to return 1 in H2. Thus he transformed:
+P:Q:R => +Q:0:P!R , (i.e., +Q:0:~Q , i.e., +Q(x)).

When H2 detects H's output via $$H(A,A) let us consider three possibilities as to what H2 is programmed to do for each value, 1 and 0. That is, there is a family of nine programs into which H could be transformed,

depending on what H2 does when $$H(A,A) equals 1 or 0. Then +P:Q:R becomes each of the following nine requirements:

= 1	= 0	require	equals	thus
quit 1	quit 1	P!Q:0:R	~R:0:R	~R(x)
quit 1	quit 0	P:Q:R		
quit 1	loop	P:0:Q!R	P:0:~P	P(x)
quit 0	quit 1	Q:P:R		
quit 0	quit 0	0:P!Q:R	0:~R:R	
quit 0	loop	0:P:Q!R	0:P:~P	
loop	quit 1	Q:0:P!R	Q:0:~Q	Q(x)
loop	quit 0	0:Q:P!R	0:Q:~Q	
loop	loop	0:0:P!Q!R	0:0:1	~TRUE(x)

Let us define function R9 as: For Turing Machine m, and AY, AN each either Y, N or L, R9(m, AY, AN) is the Turing Machine created when each halt YES in m is changed to halt YES, halt NO or a LOOP, depending on AY being Y, N or L, respectively, and each halt NO is changed likewise depending on AN. Turing's proof uses the fact that if m solves P:Q:R then R9(m,"L","Y") solves Q:0:~Q. That is, ("L","Y") in R9 accomplishes P:Q:R => Q:0:~Q.

Thus +HALT(I,I):~HALT(I,I):0 =>
+~HALT(I,I):0:HALT(I,I)and
R9($T(H),"L","Y")=$T(H2).

Turing's proof then proceeds to show that ~HALT(I,I):0:HALT(I,I), (i.e., -~HALT(x,x)). This is done by applying the program that solves ~HALT(I,I):0:HALT(I,I), (i.e., H2, to itself, and showing that it can neither halt nor loop).

To show that a particular P:Q:R cannot be solved, we consider the three possible results that can occur when it is applied to itself: halt YES, halt NO or Loop. The inconsistency occurs when the fact of the result (Yes, No, Loop) implies how it will halt (or not) according to its own definition. In Turing's case, a result of Yes would mean that we Loop (since HALT(I,I) is true), No implies Loop (still HALT(I,I) is true), and Loop implies Yes (~HALT(I,I) is true). All three are inconsistent.

Only if the predicates in P:Q:R involve relations YES, NO, HALT and LOOP can we make a new conclusion about how a program will halt on itself based on how it will halt on itself. We first consider if YES(I,I) where I is the program purporting to solve P:Q:R. If YES(I,I)logically implies P then we are okay. If YES(I,I) implies Q or R, (i.e., ~P), then we have an inconsistency. The conclusion is that YES(I,I) is false. When we also prove that both NO(I,I) and LOOP(I,I) are false, we have an inconsistency for the entire Turing Machine. It cannot exist.

Let us define symbols Y, N, L to represent YES(I,I), NO(I,I) and LOOP(I,I), respectively. When P, Q and R are functions of Y, N, L then there is the possibility that the above requirement is met for the unsolvability of P:Q:R.

Any predicate that depends only on (is a function of) Y, N, L will be defined by the result of running m against m. This result is either YES, NO or LOOP. That is, exactly one of Y, N, L is true. The predicate is defined by whether it holds for each result Y, N, L. There are eight such predicates, with specific expressions, as follows.

#	Y	N	L	expression	name
1	0	0	0	0	~TRUE
2	0	0	1	L	LOOP
3	0	1	0	N	NO
4	0	1	1	~Y	~YES
5	1	0	0	Y	YES
6	1	0	1	~N	~NO
7	1	1	0	Y!N	HALT
8	1	1	1	1	TRUE

We generate -P:Q:R in five steps:

1. Consider different values for predicates P, Q, R.
2. Consider transformations of P:Q:R via the nine functions in R9.
3. Consider the three possible results of running m against m , viz. Y, N, L.
4. What is the result of running m against m ?
   a. We are considering a specific result in step 3.
   b. Which of P, Q, R is true for this result? (P,Q,R mean a result of Y,N,L, respectively.)
   c. Is the result consistent ? That is, are (a.) and (b.) consistent ?
5. If all three possible results are inconsistent, then the requirements in step 2, and thus also step 1, are unsolvable.

An M routine, TURING, to perform this is listed below (see Figure 1). If you DO CHECKSUM^TURING, you get:

```
43726*  300  219732  61880  750  120684*
112637 1379744 1746 320 723910* 1775724
67301  448  1432920*  1960464  734587
```

```
3217266  1913756*  3245080*  672  3923436*
77418   3333504   800   3233126*   4692627
3532284  2508384  1045920*  992  7451200*
1056  6999750*   4857580  6398316  5488765
7372  809328*   2726000  1704739  160860  =
75967104
```

The first 42 numbers are the checksums for the 42 lines in routine TURING. The last number (after "=") is the grand total for the routine. An asterisk by a number means that line has a tag in the routine.

If you DO TP^TURING, the output is:

Can't solve:

```
1. -0:~HALT
2. -0:~YES
3. -0:~NO
4. -~HALT:0
5. -~HALT:NO
6. -~HALT:YES
7. -~HALT:HALT
8. -NO:~HALT
9. -NO:~NO
10. -~YES:0
11. -~YES:YES
12. -YES:~HALT
13. -YES:~YES
14. -~NO:0
15. -~NO:NO
16. -HALT:~HALT
```

Can solve:

```
1. +0:0
2. +0:NO
3. +0:YES
4. +0:HALT
5. +0:1
6. +NO:0
```

```
TURING ; * '98 MTA Meeting paper
       ;
       ;  mexpr(1-8) = M expression f(Y,N) = {0,1}
       ;            (0) = Name
       ;
TP     ;  Theorem-Prover
       K  D INIT K (mexpr)
       F CAN=0,1 D CAN ;  Print solvable & unsolvable requirements
       Q

CAN    W !!,"Can" W:'CAN "'t" W " solve:" S CT=0
       F PNM=1:1:8 F QNM=1:1:8 D PQR ;   values for P:Q (implicit R)
       W !," - - -" Q

PQR    ; see if mexpr(PNM):mexpr(QNM) is solvable
       F RES="Y","N","L" D RESXY I @mexpr(PNM),@mexpr(QNM) G INC
       S SOLVE=1 ;  no PNM,QNM overlap
       F AY="Y","N","L" F AN="Y","N","L" D SOLVE G TRY:'SOLVE ; 9 transforms
TRY    D PRINT:CAN=SOLVE ;  Print if Can/Can't solve
INC    Q  ;  PNM and QNM overlap - can't both Halt YES and Halt NO

PRINT  ; AY+AN transformation has no consistent value for <m,m>
       S CT=CT+1
       W !,$J(CT,5),". ",$E("-+",CAN+1),mexpr(PNM,0),":",mexpr(QNM,0) Q

SOLVE  ; apply R9(mexpr(PNM):mexpr(QNM),AY,AN) => E("Y"):E("N")
       F E="Y","N" S E(E)=mexpr(AY=E*PNM)_"!"_mexpr(AN=E*QNM) ;  create E(*)
       F RES="Y","N","L" D RESXY I @(E("Y")_"=Y"),@(E("N")_"=N") G OKRES
       S SOLVE=0 ;  No consistent result for <m,m>
OKRES  Q  ;  <m,m> could be RES

RESXY  S Y=(RES="Y"),N=(RES="N") Q  ;  translate Result into Y,N for mexpr(*)

INIT   ; : mexpr(*)  < Initialize universe of primitive predicates >
       S ALL="0=0,'(Y!N)=~HALT,N=NO,'Y=~YES,Y=YES,'N=~NO,Y!N=HALT,1=1"
       K mexpr S mexpr(0)=0 ;  to get M expr or 0 from mexpr(boolean*#)
       F A=1:1:8 S B=$P(ALL,",",A),mexpr(A)=$P(B,"="),mexpr(A,0)=$P(B,"=",2)
       Q
CHECKSUM    S A=0 F B=1:1:42 D
        . S C=$T(+B),D=0 F E=1:1:$L(C) S D=B*E*$A(C,E)+D
        . S A=A+D W $J(D,9),$E("* ",$E(C)=" "+1)
       W " = ",A Q
```

Figure 1

```
 7. +NO:YES
 8. +YES:0
 9. +YES:NO
10. +HALT:0
11. +1:0
```

A requirement of the form `P:~P:0` means that `P(I)`, (i.e., the set `P` is recursive). Requirement `P:0:~P` means that `P(x)`, (i.e., set `P` is recursively enumerable). These formats occur in unsolvable requirements 4, 7, 9, 10, 11, 13, 14, 15 and 16. Thus these are equivalent to:

```
1. -~HALT(x,x)
2. -~HALT(I,I)
3. -NO(I,I)
4. -~YES(x,x)
5. -~YES(I,I)
6. -YES(I,I)
7. -~NO(x,x)
8. -~NO(I,I)
9. -HALT(I,I)
```

Since `P(I,J) => P(I,I)` by substitution, then `-P(I,I)=>-P(I,J)`. Thus 2, 3, 5, 6, 8, 9 above easily imply `-~HALT(I,J)`, etc. Many of these are theorems familiar to the Theory of Computation, informally proven by hand in the past. For example, `HALT(I,J)`, `HALT(I,I)` and `YES(I,J)` are often called the halting, self-applicability, and membership problems [2,3,4].

Routine TURING also lists requirements for which it did not find an inconsistency, stating "Can solve." Can we really solve all of these? Looking through this list we see still more of our friends from the Theory of Computation:

```
1. +~TRUE(X)
2. +NO(x,x)
3. +YES(x,x)
4. +HALT(x,x)
5. +TRUE(x)
```

Number 9 `+YES(I,I):NO(I,I)` is the so-called Universal Turing Machine. The key to being able to solve all of these is the fact that a Turing Machine can be constructed that interprets the running of another Turing Machine (just as interpreters are written for computer languages). **M**

References

1. Martin Davis, (ed.), *The Undecidable* (New York: Raven Press, 1965.)
2. Dexter C. Kozen, Automata and Computability Springer (1997).

3. Marvin L. Minsky, *Computation: Finite and Infinite Machines* (New York: Prentice-Hall Inc., 1967).

4. Rozenberg, Grzegorz and Salomaa, Arto *Cornerstones of Undecidability* (New York: Prentice-Hall Inc., 1994).

5. Jean van Heijenoort, *From Frege to Godel: A Source Book in Mathematical Logic 1879-1931* (Cambridge: Harvard University Press 1967).

6. Charles Volkstorf, "Debugger: A MUMPS Program that Detects Errors in MUMPS Programs," *MUG Quarterly*, 12, 2, Summer 1982.