The Recursion Theorem

Robert M. Keller

Harvey Mudd College

11 April 2005

A "Simpler" Proof that the Halting Problem is Not Computable

- The following function is not computable:

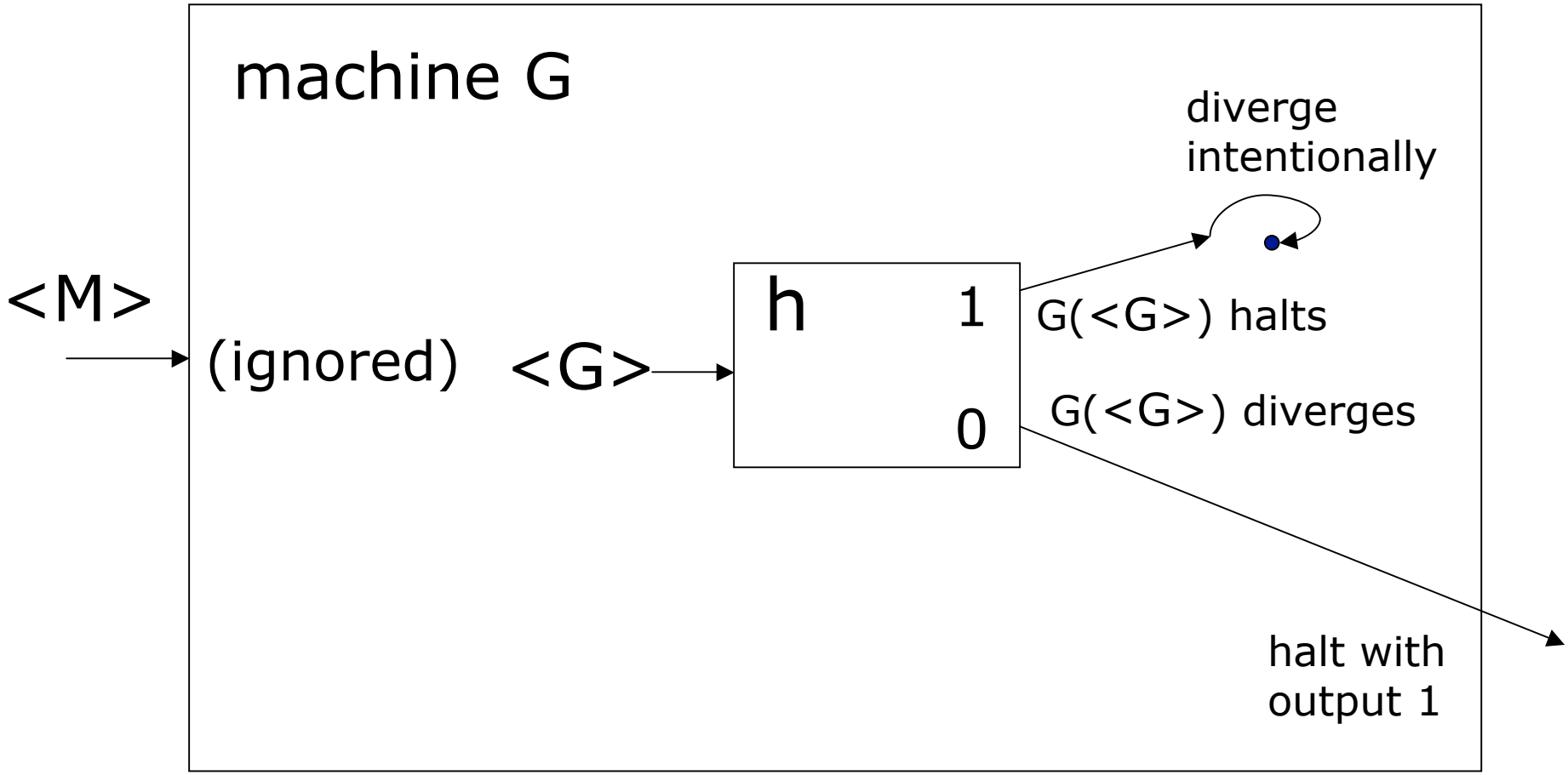
$$h(\langle M \rangle) = \begin{cases} 1 \text{ if } M \text{ halts on input } \langle M \rangle \\ 0 \text{ otherwise} \end{cases}$$

- Proof: Suppose h were computable. Then so would the following function **g** computed by machine **G**:

- G with input <M> (ignores <M> and) computes h(<G>).
  If 1 is returned from h(<G>), G **diverges** intentionally; otherwise G returns 1.

- So g(<G>) diverges iff (from h) G halts on <G>, a contradiction.

The machine G always gets it wrong, so the hypothesized computability of h is false.



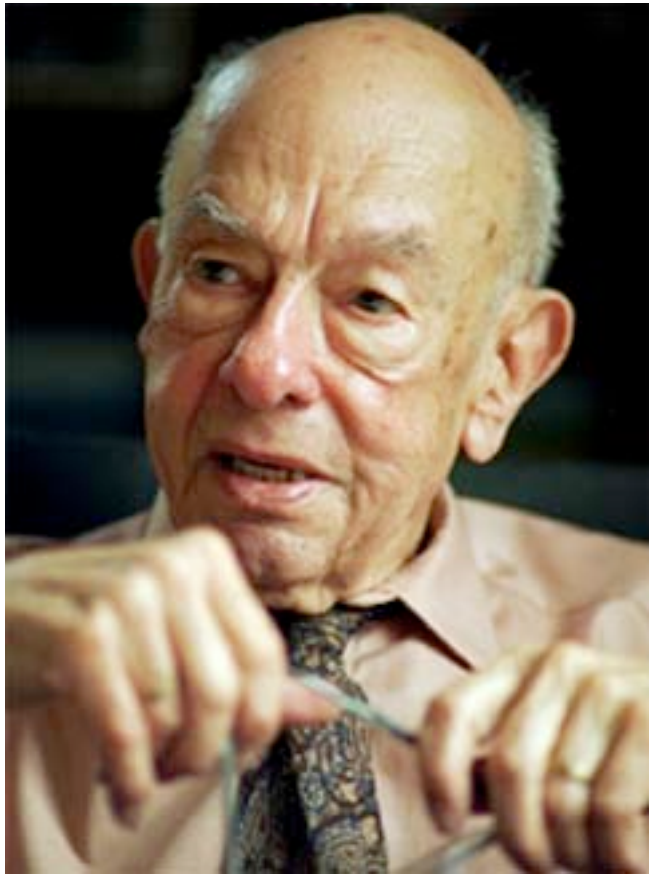
Self-Reference

- In the previous proof, we constructed a machine G that used its own description <G>.
- Is this allowed?
- Can a machine have access to its own description?
- Or does a TM A printing the description of TM B require that A be strictly larger or “more powerful” than B?

There are Turing machines that can print their own description

- At first this might sound impossible.
- Machines that can print their own description have value in establishing certain theoretical results.
- Such machines (and programs in various languages) have recently been called “Quines” after Willard Van Orman Quine, a famous logician who wrote on self-referential paradoxes.

Willard Van Orman Quine (1908 - 2000)



A Quine in Java

Modified from one by
Levente Sántha (http://www.knet.ro/lsantha/)
(This is actually all one line.)

```
class Quine {public static void
main(String[]dummy){char quote =
34;System.out.print(tape + quote + tape + quote
+ ';' + '}');} static String tape = "class
Quine {public static void
main(String[]dummy){char quote =
34;System.out.print(tape + quote + tape + quote
+ ';' + '}');} static String tape = ";}
```

Difficulties Constructing Quines

- One difficulty is **quoting**. To get a literal symbol to print, the normal way is to quote it. But that means that the quote marks have to be printed too. And of course, they will have to be quoted, which means that escape characters may be needed. And the escape characters themselves will have to be printed.

- A workaround for this is to use **numeric codes**, that print as characters in lieu of the actual literal symbols. The numeric codes don’t normally have to be quoted.

Quines in C and C++ (authors unknown)

C Quine using numeric codes:

```
char f[] =
"char f[] =%c%c%s%c;%cmain() {printf(f,10,34,f,34,10,10);}%c";
main() {printf(f,10,34,f,34,10,10);}
```

This C++ Quine does not use numeric codes:

```
#include <iostream>
#define a(b) std::cout<<"#include <iostream>\n#define a(b) "<<#b<<"\nmain(){a("<<#b<<");}"
main(){a(std::cout<<"#include <iostream>\n#define a(b) "<<#b<<"\nmain(){a("<<#b<<");}");}
```

Example: A rex Quine
constructed by a Pomona College Student

```
a="\\";aa="a";
b="\"";bb="b";
c="=";cc="c";
d="print(

                aa,c,      b,    a,a,b,     f,
               aa,aa,     c,b,aa,b,f,g,bb,c,b,
              a,b,b,f    ,bb,bb,c,b,bb,b,f,g,
             cc,c,b,c   ,b,f,   cc,cc,c,b,cc,
            b,    f,g  ,dd            ,c,b,
           d,     b,f   ,              g,g,
          dd,     dd,                  c,b,
         dd,      b,f                 ,g,ee
        ,c,b      ,e,                 b,f,
        ee,       ee,                 c,b,
       ee,b,f,     g,ff,c,            b,f,
       b,f,ff,ff,c,b,ff,b,f,      g,gg,
                c,b,a          ,nn,b,
                 f,gg         ,gg,c,b,
                 gg,b,f,       g,nn,nn,c
                ,b,nn,b,      f,g,g,d,g);";
```

```
dd="d";
e=")";ee="e";
f=";";ff="f";
g="\n";gg="g";
nn="n";

print(

                aa,c,      b,    a,a,b,     f,
               aa,aa,     c,b,aa,b,f,g,bb,c,b,
              a,b,b,f    ,bb,bb,c,b,bb,b,f,g,
             cc,c,b,c   ,b,f,   cc,cc,c,b,cc,
            b,    f,g  ,dd            ,c,b,
           d,     b,f   ,              g,g,
          dd,     dd,                  c,b,
         dd,      b,f                 ,g,ee
        ,c,b      ,e,                 b,f,
        ee,       ee,                 c,b,
       ee,b,f,     g,ff,c,            b,f,
       b,f,ff,ff,c,b,ff,b,f,      g,gg,
                c,b,a          ,nn,b,
                 f,gg         ,gg,c,b,
                 gg,b,f,       g,nn,nn,c
                ,b,nn,b,      f,g,g,d,g);
```

For many Quines in many diverse languages, see:

http://www.nyx.net/~gthompso/quine.htm

For other varieties of “signature” programs, see:

http://home.planet.nl/~faase009/Signindex.html

The General Idea of Quining

- Apply a program P to a string Q:

- Print the string below, followed by the string in quotes.
- "Print the string below, followed by the string in quotes."

- P reproduces its argument Q followed by a quoted version of Q,

- while Q is, or produces the text of P

- Very roughly: P Q ⇒ print(Q(), '"', Q(), '"') ⇒ P Q

Engineering a Java Quine

1. Start with a program for P:

+ is string concatenation
34 is the ascii code for a double quote

```
class Quine
{
public static void main(String[]dummy)
{
char quote = 34;
System.out.print(tape + quote + tape + quote);
}

static String tape = "trial tape contents";
}
```

Engineering a Java Quine

2. Replace the tape contents with the text of the program itself, stopping where the tape contents string would appear:

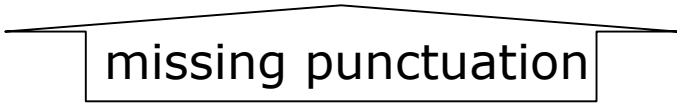
```
class Quine
{
public static void main(String[]dummy)
{
char quote = 34;
System.out.print(tape + quote + tape + quote);
}

static String tape = "class Quine {public static void
main(String[]dummy){char quote = 34;System.out.print(tape + quote +
tape + quote);} static String tape = ";
}
```

Engineering a Java Quine

3. Run the program and notice that, except for spacing, the result is nearly the program itself, with some small differences:

```
class Quine {public static void
main(String[]dummy){char quote =
34;System.out.print(tape + quote + tape +
quote);} static String tape = "class Quine
{public static void main(String[]dummy){char
quote = 34;System.out.print(tape + quote +
tape + quote);} static String tape = “
```



Engineering a Java Quine

4. Add in the punctuation chars needed to make up the difference (shown here in bold red). Eliminate line breaks.

```
class Quine {public static void
main(String[]dummy){char quote =
34;System.out.print(tape + quote + tape +
quote + ';' + '}');} static String tape =
"class Quine {public static void main
(String[]dummy){char quote =
34;System.out.print(tape + quote + tape +
quote + ';' + '}');} static String tape = ";}
```

Engineering a Java Quine

5. Run the revised program and diff, to see that the result printed is identical to the program itself. (This is really all on one line.)

```
class Quine {public static void
main(String[]dummy){char quote =
34;System.out.print(tape + quote + tape +
quote + ';' + '}');} static String tape =
"class Quine {public static void main
(String[]dummy){char quote =
34;System.out.print(tape + quote + tape +
quote + ';' + '}');} static String tape = ";}
```

Pieces of the Java Quine

```
class Quine {public static void main(String[]dummy){char
quote = 34;System.out.print(tape + quote + tape + quote +
';' + '}');}
```

P

```
static String tape = "class Quine {public static void
main (String[]dummy){char quote =
34;System.out.print(tape + quote + tape + quote + ';'
+ '}');} static String tape = ”;
```

Q

```
 }
```

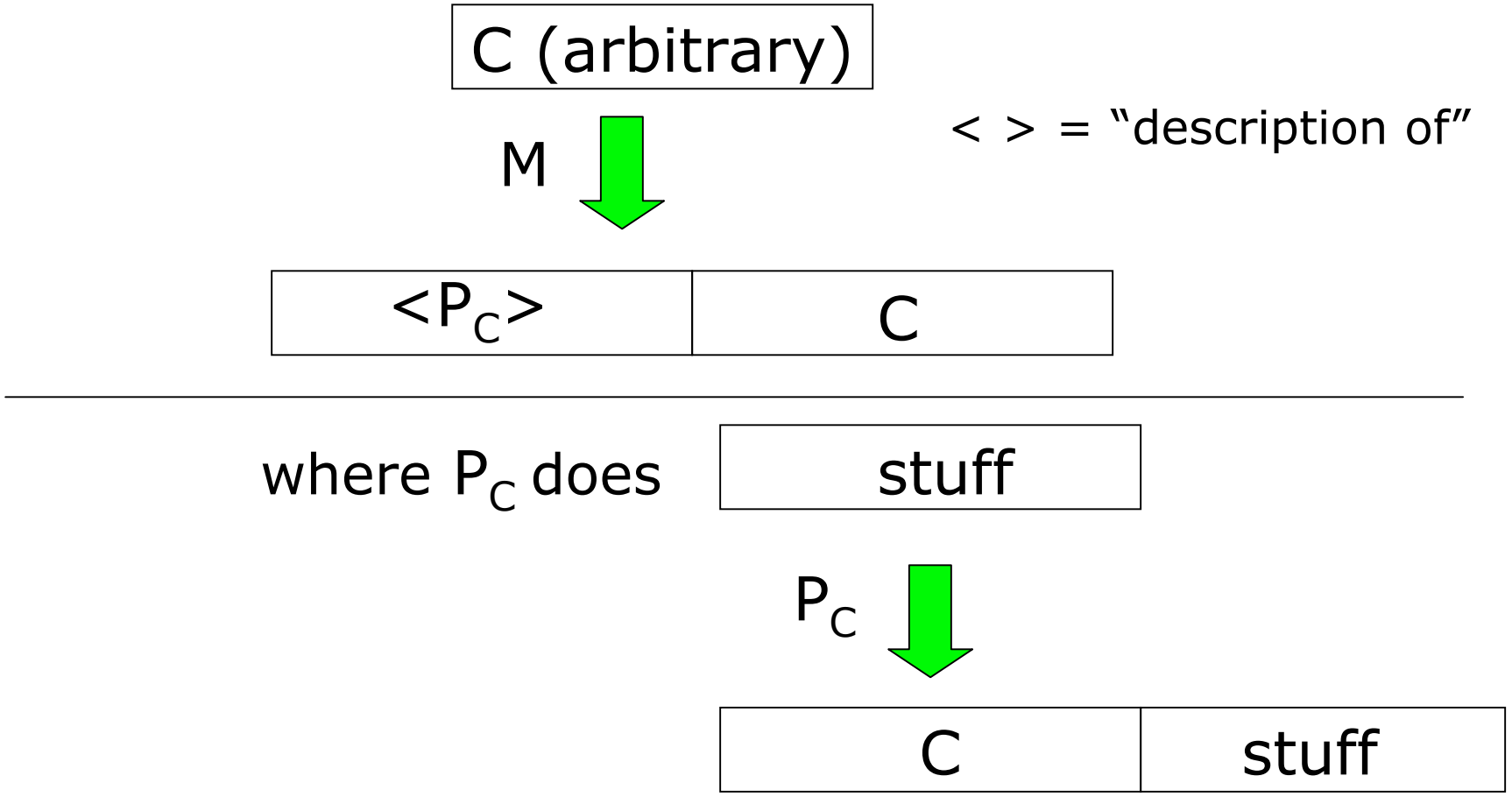
P

Unfortunate Application of Quines

Turing Machine Quine

- Let C be the contents of an arbitrary region of tape (say one delimited by blanks).
- **First** construct **M**, a machine that:
- reads any tape contents C, while printing the **description** of a machine P_C that will print C,
- followed by printing C itself.
- Use q_0 to name the initial state of the machine P_C.
- So M, with input C, prints P_C then C.

Behavior of M

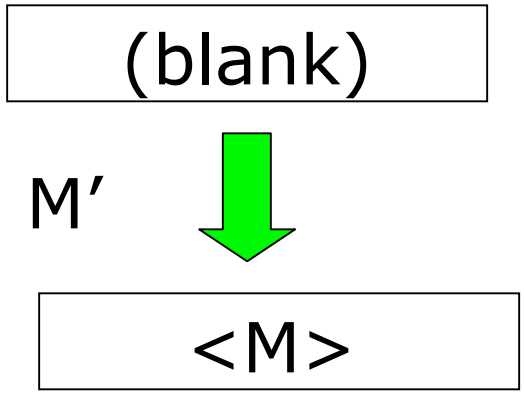


Turing Machine Quine, continued

- Next construct M’ a machine that can print the description of the **fixed** machine M on an initially blank tape.

- Then M’ will transfer control to a state named q_0 which is not otherwise in M’ (but as we know, is the name of the initial state of P_C).

Behavior of M’



where M is as on the previous diagram.

Turing Machine Quine, continued

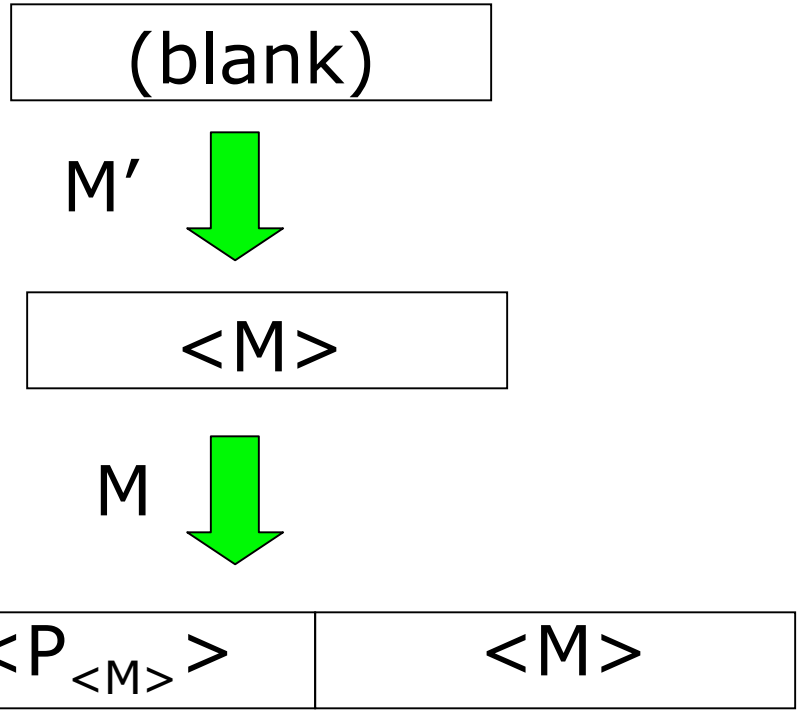
- Consider the combined program M’ M.
- This sequence will:
  - Print <M>, the description of M.
  - Print the description of
    - (a machine that prints <M>) followed by <M>

describes M’

- So the description written will be <M’><M> = <M’ M>.

- Hence this machine prints its own description.

Summary of M′ M



But $P_{<M>}$ ***is*** M′, so the final tape is <M′ M>.

The Recursion Theorem (Kleene 1938)

Informal English Statement:

- A Turing machine can have access to its own description, and use it in an argument to its own computation.

The Recursion Theorem

- Let f: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ be any partial function computable by a Turing machine.

- Then there is a Turing machine R that computes the function r defined by:

$$\forall x \in \Sigma^* \quad r(x) = f(\langle R \rangle, x)$$

Proof of The Recursion Theorem

- A two-argument Turing machine is one where there is an understood separator (such as a blank) that separates the two arguments.
- The proof is analogous to showing the existence of a self-printing machine.
- Notation: For any string w, define $\mathbf{P_w}$ to be a machine that prints w.

Proof of The Recursion Theorem (from Michael Sipser, 1997)

- Let f: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ be computed by a TM **F**.
- **R** is constructed as A B F (similar to in the self-printing machine).
- Let B be a machine that computes b, where for any w, b(w) = $\langle P_w \rangle$, the description of a machine printing w.
- A will be $P_{\langle B\ F \rangle}$, a machine that prints <B F>.
- So after A runs on input x, <B F> will be on the tape *before* the original input x.
- Then B runs, producing $\langle P_{\langle B\ F \rangle} \rangle = \langle A \rangle$, before <B F>, resulting in <A B F> = <R>.
- Then F runs on the tape, producing f(<R>, x).

Partial Recursive Function Version of The Recursion Theorem (Kleene 1938)

- (In recursive function theory, descriptions are replaced by numeric indices.)

- For any partial recursive function ψ of n+1 variables, there is a number e, such that
$(\forall\ x_1, x_2, \ldots x_n)$

$$\varphi^n_e(x_1, x_2, \ldots x_n) = \psi(e, x_1, x_2, \ldots x_n)$$

where $\varphi^n_1, \varphi^n_2, \ldots$ is an effective enumeration of all n-argument partial recursive functions. (Being the number of φ^n_e, e is analogous to a description of ψ.

Fixed-Point Theorem (TM version)

- For any partial function ψ computable by a Turing machine,

  there is a Turing machine F such that $\psi(\langle F \rangle)$ is equivalent to F,

  in the sense that both F and $\psi(\langle F \rangle)$ compute the same partial function.

Fixed-Point Theorem (PRF version)

- For any partial recursive function ψ, there is an index f such that

$$\varphi_{\psi(f)} = \varphi_f$$

Proof of the Fixed Point Theorem
(TM Version, from Sipser 1997)

- Let ψ: $\Sigma^* \rightarrow \Sigma^*$ be computed by a TM. We want to construct an F so that ψ(<F>) is a TM equivalent to F.

- F is defined to behave as follows on input x:
  - Get the description of **this** machine <F>. (using recursion thm)
  - Compute ψ(<F>) and interpreting it as the description of a Turing machine, say G, and apply G to x.

- Evidently F and G are equivalent, as F gives the same result on any input x that G gives a result.

- So the function computed by F is the same as that computed by G.

- Thus F is the fixed point of ψ.

An Application of the Fixed Point Theorem (Sipser, 1997)

- The **length** of the description of a machine <M> is the number of symbols in <M>.
- M is called **minimal** if there is no equivalent machine having a shorter description.
- **Theorem**:

  The language { <M> | M is a minimal TM }
  is not recursively-enumerable.

- In other words, there is no way to recognize whether a given description is minimal.

Proof

- Assume that { <M> | M is a minimal TM } is enumerated by a Turing machine E.
- Construct the following TM, call it C, which, on input w:
  - Obtain the description <C> of this machine.
  - Run E on successive inputs {0, 1, 2, 3, …} until a machine D appears such that <D> is **longer** than <C>.
  - Run D on w.
- C is equivalent to D by construction.
- But C is shorter than D, therefore D cannot be minimal after all. This contradicts the assumption that E enumerates only minimal machines.

Implications

- There is no algorithm for computing a shortest program equivalent to a given program.

- In fact, no algorithm can even recognize the shortest program when it sees it.