The Recursion Theorem

Robert M. Keller
Harvey Mudd College
April 2010

What is this?
- A fundamental result having to do with computability and programming languages.
- Another technique that can be used to get further undecidability results.
- It was introduced as a theorem by Kleene in 1938.

Recursion Theorem: Informal Statement
- A program can have access to its own description (code).

Another undecidability proof for \( A_{TM} \)
- This proof uses self-reference rather than diagonalization, as in our first proof.
- Suppose there is a TM that decides \( A_{TM} \) to get contradiction.

Another undecidability proof for \( A_{TM} \)
- Suppose \( H \) is a TM that decides \( A_{TM} \).
- Construct a machine \( N \) that behaves as follows:
  - With input \( x \), run \( H \) on \( \langle N, x \rangle \).
  - If \( H \) accepts, reject. If it rejects, accept.
  - Note here that \( N \) is a description of this very machine.
- What will \( N \) do with input \( \langle N \rangle \)?
  - If \( N \) accepts \( \langle N \rangle \), then \( H \) on \( \langle N, \langle N \rangle \rangle \) rejects, indicating \( N \) does not accept \( \langle N \rangle \).
  - If \( N \) does not accept \( \langle N \rangle \), then \( H \) on \( \langle N, \langle N \rangle \rangle \) accepts, indicating \( N \) accepts \( \langle N \rangle \).
  - Either way, we contradict the supposition of such an \( H \).
**Functional description**

- $H(<M, x>) = M(x) \ ? \ true : false$
- $N(x) = \neg H(<N, x>)$
- So $N(<N>) = \neg H(<N, <N>>)$
- Constr. of $N$
- Def. of $H$
- Meaning of $\neg$

(unless $N<N>$ never halts, but $N$ must always halt if $M$ does)

**What is key in the previous proof?**

- It relied on the ability of a machine to use its own description inside its own program.

- Is this strange?
  - An interpreter could use its own source code, for example, and interpret that code.
  - Ok, but is it strange for TMs?

**Self-Printing Machines**

- Even if a machine is not given a handle to its own code on its tape at the outset, there are ways for it to construct it.

- Such programs are now called "Quines". (Would Quine like this?)

**A Java Quine (all one line. 34 is ")**

```java
class Q{public static void main(String[]v){char c=34;System.out.print(s+c+s+c+';')}static String s="class Q{public static void main(String[]v){char c=34;System.out.print(s+c+s+c+';')}static String s=";"}main(){}}
```

% javac Q.java
% java Q

source: http://www.knet.ro/lsantha/

**Quines in C and C++ (authors unknown)**

C Quine using numeric codes:

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

This C++ Quine does not use numeric codes:

```cpp
#include <iostream>
define a(b) std::cout<<"<include "iostream>"define a(b) <<"<include "iostream>"main()<<"a">"<<"<include "iostream>"define a(b) "<<"<include "iostream>"main()<<"a">"<<"<include "iostream>"define a(b) "<<"<include "iostream>"main()<<"a">"<<"<include "iostream>"define a(b) "<<"<include "iostream>"main()<<"a">"<<"<include "iostream>"define a(b) "<<"<include "iostream>"main()
```
Example: A rex Quine constructed by a Pomona College Student

```

```

continued next col.

Applications of Quines

- Entertainment of self and others
- Computer viruses, worms, and other forms of mal-ware
  - To protect against these, it is important to know their characteristics and methods of operation.
- Artificial life
  - cf. von Neuman: "Theory of Self-Reproducing Automata"

Recursion Theorem Formalized

- If R is a Turing machine computing a binary function R(A, B), then there is a Turing machine S computing a unary function such that:
  \[ S(A) = R(A, <S>) \]
  where \(<S>\) is the description of S itself.

From Programming Languages

- Compute a recursively-defined function \textit{without actually using recursion.}
  - This is not so hard if we allow higher-order functions (functions that take functions as arguments and return functions as results). These are sometimes called "functionals".

Example

- Factorial:
  \[ \text{fac}(N) = N < 2 \ ? 1 : N \ast \text{fac}(N-1) \]
  - How to do this \textit{without} recursion?

Functionalize the definition

- "Factorial" functional
  \[ f(G)(N) = N < 2 \ ? 1 : N \ast (G(G)(N-1)) \]
  - Notice the above definition is \textit{not recursive}.
  - G could be any function argument.
Functionalize the definition

- \( f(G)(N) = N < 2 \ ? \ 1 : N^4(G(G)(N-1)) \)
- \( G \) could be any function argument.
- \( f(f) \) makes sense:
  - \( f(f)(N) = N < 2 \ ? \ 1 : N^4(f(f)(N-1)) \)
  - So \( f(f) \) achieves the same effect as \text{fac}.
- We might say \( f(f) \) "is" \text{fac}?
  - It is more correct to say "\text{fac} is a fixed point of \text{f}" (\text{fac} satisfies the functional equation when substituted for \( G \)).
  - In fact (oops), \text{fac} is the least fixed point of functional \text{f}.

Least Fixed Point?

- Least in this case means "least defined".
- That is, it is the fixed point that makes the fewest assumptions consistent with the definition of \text{f}.
- In the case of \text{f}, \text{fac} is the only fixed point.
- In other cases, there can be more than one, with varying degrees of defined-ness.

Example Realization (in rex)

- 1 rex > \( f(G)(N) = N < 2 \ ? \ 1 : N^4(G(G)(N-1)) \);
- 2 rex > \( f(f)(10) \);
- 3628800

An Application of the Recursion Theorem (Sipser)

- 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 \( \{ <M> \mid M \text{ is a minimal TM} \} \) is not recognizable.

Proof

- Assume that \( L = \{ <M> \mid M \text{ is a minimal TM} \} \) is recognizable. Then \( L \) is enumerated by some Turing machine \( E \).
- Construct the following TM, call it \( C \), which, on input \( w \):
  - Obtain the description \( <C> \) of this machine.
  - Using \( E \), begin enumerating \( L \) until a machine \( D \) appears such that \( <D> \) is longer than \( <C> \). (This must happen.)
  - Behave as \( D \) on \( w \).
- \( C \) is equivalent to \( D \) by construction.
- But \( <D> \) is longer than \( <C> \), therefore \( D \) cannot be minimal after all. It shouldn’t be in the enumeration. This contradicts the assumption that \( E \) enumerates only minimal machines.