Machines that know their own name: Applications of the Recursion Theorem

Bernard Anderson

Gordon State College

November 19, 2012

Introduction

Computability Theory

- Computability Theory (also called Recursion Theory) is a branch of mathematical logic.
- We study sets of numbers, looking at their properties, patterns, and things in common with other sets that share these properties

Computability Theory (continued)

- We also study their computational power, what other sets of numbers to they compute (and which compute them)
- Finally we look at the sets as a single structure whose shape is given by this computability relation
- Fundamental open question: Is this structure rigid, can we preserve the shape if we move the sets around?

Today's talk

In this talk we will learn about one of the main theorems of introductory Computability Theory, and some of its consequences

Today's talk

In this talk we will learn about one of the main theorems of introductory Computability Theory, and some of its consequences

But first, there is a story…

Computability

Definitions of computable

- Turing machines
- Abacus machines
- Recursive functions

Computability

Definitions of computable

- Turing machines
- Abacus machines
- Recursive functions
- All definitions are equivalent (each implies the others)

Computability

Church's Thesis

- Our intuitive definition of Computable is correct.
- If we can describe an effective procedure to calculate a function, then it is computable.
- This hypothesis has held for well over half a century of research in mathematics and computer science.

Computability

Definition

We say a function is computable if it can be calculated by a sufficiently powerful supercomputer using arbitrarily large finite amounts of time and memory space.

Programs

Computer Programs

- We view computer programs as strings of symbols
- Many, like “alsdjfjpiel3fne!lneij;tgieja” don’t do anything.
- Others, like “Input x, Output $x + 1$” work as intended.

Programs

Definition

We say a program is total if for every input the program comes to a halt and provides meaningful output.

Enumeration

Listing programs

- We want to find a way to list off all possible computer programs.
- We let P_n denote the n-th program in our list.
- One example is...

Enumeration

One possible list

P_1: a

P_2: b

P_3: aa

P_4: c

P_5: ab

P_6: ba

P_7: aaa

$\vdots$

$P_{9738928}$: Input x, Output $x + 1$

$\vdots$

Other lists

- Clearly, there are better lists available.
- However, in Computability Theory, which list we use doesn't matter
- We only need to make sure the list obeys the following properties:

Enumeration

Properties for lists

1. It must list every computable function.
2. It must be computable to find P_n given n.
3. It must be computable to find n given P_n.

We call such lists acceptable and assume we are working with some fixed acceptable list.

Total lists

Question

Why not only list the total programs?

Total lists

Question

Why not only list the total programs?

Answer

It is impossible for such a list to exist.

Total lists

Proof

Suppose $P_1, P_2, P_3, \ldots$ is an acceptable list of total programs.

We define a new program:

$$\Phi(x) = P_x(x) + 1$$

Φ is total so it must be on the list. Let Φ be P_n.

Then $\Phi(n) = P_n(n) + 1 = \Phi(n) + 1$ for a contradiction.

We conclude no such list exists. $\square$

Recursion Theorem

Main result

We now come to the main result of this talk

But first, back to our story...

Recursion Theorem

Theorem

Let $f(x, y)$ be a computable function. Then there is a n such that $P_n(x) = f(x, n)$

Examples

- $P_n(x) = n$ (for any x)
- $P_n(x) = x + n$
- $P_n(x)$ simultaneously runs all $P_m(x)$ with $m > n$ and outputs the first calculation that halts

Recursion Theorem

Notes

- This holds for any acceptable ordering.

Notes

- This holds for any acceptable ordering.
- There are infinitely many such n.
- They can be found effectively.

Recursion Theorem

Counterexample attempt

- We can ask, what goes wrong if we try to create a listing where the Recursion Theorem doesn't hold?
- The idea is to look at the output of each program and assign it a spot on the list where it doesn't satisfy the theorem.
- This doesn't work because some programs aren't total, and others are total but take arbitrarily long to run.
- As proved earlier, an acceptable list of only total functions doesn't exist.

Use in research

Using the Recursion Theorem

- We sometimes want to show a function with excessive predictive power does not exist.
- It seems obvious that the predictions can't all be correct, but it is hard to find one that is wrong.
- The Recursion Theorem is often useful in finding counterexamples in these cases.

Use in research

Using the Recursion Theorem

- We sometimes want to show a function with excessive predictive power does not exist.
- It seems obvious that the predictions can't all be correct, but it is hard to find one that is wrong.
- The Recursion Theorem is often useful in finding counterexamples in these cases.
- We will see two examples from my own research (highly simplified).
- The theorem can also be used for several other purposes

Use in research

Definitions

- We let $P_n^\sigma(x)$ be program n run with input x where the program is allowed to use information from σ
- Think of σ as information on a flash drive plugged into our computer.

Use in research

First example

- We want to show the program Φ and computable function f don't exist
- Φ has the property that it can predict $P_x^\sigma(x)$ only looking at the first $f(x)$ bits of information in σ.
- It seems obvious they don't exist, but hard to prove.
- We don't know the value of f for the program we are writing, until we finish writing it.

Use in research

Solution

- We use the Recursion Theorem.
- We define $P_n^\sigma(x)$ to be zero if the $f(n)+1$ piece of information in σ is zero.
- Otherwise we have program $P_n^\sigma(x)$ never halt.
- The behavior of P_n^σ depends entirely on bit $f(n)+1$ of σ.
- Hence Φ cannot predict it by looking only at the first $f(n)$ bits of σ.

Use in research

Second example

- We are trying to show a binary function Γ does not exist.
- Γ is total, and has the property:

  For all x, if $\Gamma(x) = 1$ then $P_x(x)$ halts (with output).
- To start our counterexample, we want to find a n such that $\Gamma(n) = 0$ and $P_n(n)$ halts.

Use in research

Solution

- Using the Recursion Theorem, we define:

$$P_n(x) = \begin{cases} P_n(n) + 1 & x = n \text{ and } \Gamma(n) = 1 \\ 0 & \text{otherwise} \end{cases}$$

- If $\Gamma(n) = 1$ then $P_n(n) = P_n(n) + 1$ for a contradiction.
- Hence $\Gamma(n) = 0$. Note then $P_n(n) = 0$ and hence halts.

Proof

s-m-n Theorem

- We start to look at the proof of the Recursion Theorem. We first consider related results.
- Suppose we have a given program $\Phi(x, y)$ with two inputs. Fix some number m and define a program $\Gamma(x) = \Phi(x, m)$.
- All Γ does is run Φ with the second input chosen in advance to be m.
- Given Φ and m the program Γ is easy to describe.

Proof

s-m-n Theorem (continued)

- The s-m-n Theorem says we can effectively find the name of program Γ given m
- It holds because our listing of programs is acceptable, i.e. we can find the name when given the program.

Theorem

Let $f(x,y)$ be a computable function. Then there is a one-to-one computable function g such that $P_{g(y)}(x) = f(x,y)$.

Proof

Recursion Theorem

- The version of the Recursion Theorem that we used is slightly different than the original version below.
- We will prove the version we used follows from the original on the next slide.

Theorem (Kleene)

Let f be a computable function. Then there is a number n such that $P_n = P_{f(n)}$.

Proof

Proof of corollary

- Let $f(x,y)$ be given. We wish to show there is a n such that $P_n(x) = f(x,n)$.
- By the s-m-n Theorem, let g be such that $P_{g(y)} = f(x,y)$.
- By the Recursion Theorem, let n be such that $P_n = P_{g(n)}$.
- Then $P_n(x) = P_{g(n)}(x) = f(x,n)$. $\square$

Proof

Idea for main proof

- The proof of the Recursion Theorem is short, but notoriously difficult to memorize.
- We create a very strange "diagonal" function, d, and use it to define n.
- We then have a brief but intricate verification that n works.

Proof

Proof [3]

- Let f be computable. We want to find n such that $P_n = P_{f(n)}$.
- By the s-m-n Theorem we define $d(x)$:

$$P_{d(y)}(x) = P_{P_y(y)}(x)$$

- Let v be such that $P_v(x) = f(d(x))$. Let $n = d(v)$. Then:

$$P_n = P_{d(v)} = P_{P_v(v)} = P_{f(d(v))} = P_{f(n)} \quad \square$$

Proof

Remarks

- We note that we have an explicit computation to find n.
- It is not difficult to alter the proof to find infinitely many such n.
- It can also be expanded to handle more variables, etc.

Conclusion

Further study

- There are a lot of other interesting results in introductory Computability Theory.
- The field is still fairly new and expanding.
- The objects we study are relatively tangible.

Conclusion

References

1. B. A. Anderson. Automorphisms of the truth-table degrees are fixed on a cone. *J. Symbolic Logic*, 74(2):679–688, 2009.

2. B. A. Anderson and B. F. Csima. A bounded jump for the bounded Turing degrees. *Notre Dame J. Formal Logic*, To appear.

3. R. I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1987.

Conclusion

Thank you