An introduction to computability:

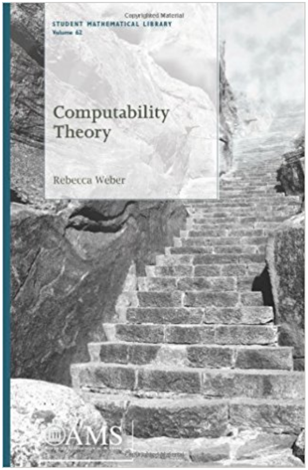
an excerpt from my DRP with Noah Hughes

Tristan Knight
tristan.knight@uconn.edu
University of Connecticut

Apr. 28, 2017

University of Connecticut
DRP Seminar

My book



If I can program my computer to execute a function, to take any input and give me the correct output, then that function should certainly be called computable.

What is computability?

- Computability is the ability to solve a problem.
  - A computable problem should have a list of steps that can be followed to solve it.
- Computability theory is the mathematical treatment of computability.
  - Mathematical interpretations of complexity, algorithms, and so forth.

What is computability?

- Computability is the ability to solve a problem.
  - A computable problem should have a list of steps that can be followed to solve it.
- Computability theory is the mathematical treatment of computability.
  - Mathematical interpretations of complexity, algorithms, and so forth.

Issue: How do we make such a general concept rigorous?

Some prior definitions

Definition: A partial function on $\mathbb{N}$ is a function whose domain is a subset of $\mathbb{N}$.
Definition: A total function on $\mathbb{N}$ is a function with domain $\mathbb{N}$.

Some prior definitions

Definition: A partial function on $\mathbb{N}$ is a function whose domain is a subset of $\mathbb{N}$.
Definition: A total function on $\mathbb{N}$ is a function with domain $\mathbb{N}$.

Definition: A partial function f halts on a natural number x if x is in the domain of f.

Turing machines

A Turing machine is the “simplest” computer, and is our first example of a definition for computable functions.

Turing machines

A Turing machine is the "simplest" computer, and is our first example of a definition for computable functions.

A Turing machine is defined in terms of the following:

- States: $\{q_0, q_1, \ldots, q_n\}$.
- Finite alphabet: e.g. $\{, 0, 1\}$.
- Infinite tape, divided into sections.
- "Read-write head" that can:
  - Read the current section.
  - Change the symbol on the tape.
  - Move left or right on the tape.
- Finite set of instructions.

Turing machine: example

Alphabet: $\{*, 0, 1\}$

States: $\{q_0, q_1\}$

Instructions:

- $\langle q_0, 0, *, q_1 \rangle$
- $\langle q_0, 1, *, q_1 \rangle$
- $\langle q_1, *, R, q_0 \rangle$

Starting tape: $\{\ldots, *, *, 1, 0, 1, 0, 1, *, *, \ldots\}$

Primitive recursion

Definition: The class of primitive recursive functions is the smallest class $\mathcal{C}$ such that

1. Constant function: $0(x) = 0$ is in $\mathcal{C}$.
2. Succesor: $S(x) = x + 1$ is in $\mathcal{C}$.
3. Projection: $P_i^n(x_1, x_2, \ldots, x_i, \ldots, x_n) = x_i$ is in $\mathcal{C}$.
4. Function composition: substitution of functions in $\mathcal{C}$ for the variables of a function in $\mathcal{C}$ produces a function in $\mathcal{C}$.
5. Recursion: if $g, h \in \mathcal{C}$, then the function f given by
   - $f(x_1, \ldots, x_n, 0) = h(x_1, \ldots, x_n)$
   - $f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$

   is in $\mathcal{C}$.

Primitive recursion (example)

We show $f(x, y) = x + y$ is primitive recursive.

Primitive recursion (example)

We show $f(x, y) = x + y$ is primitive recursive.
Note: We wil use the notation $+(x, y)$ to refer to $x + y$ as it meshes much better with the notation used in the definition of primitive recursion.

Proof.
Define $+(x, y)$ as follows:

- $+(x, 0) = P_1^1(x)$
- $+(x, y + 1) = S(P_3^3(x, y, +(x, y)))$.

By our definition of primitive recursion, $+(x, y) = x + y$ is primitive recursive. $\square$

Primitive recursion (another example)

We show $f(x, y) = x \times y$ is primitive recursive.

Primitive recursion (another example)

We show $f(x, y) = x \times y$ is primitive recursive.

Proof.

Define $\times(x, y)$ as follows:

- $\times(x, 0) = P_1^1(x)$
- $\times(x, y + 1) = +(P_3^3(x, y, \times(x, y)), P_1^3(x, y, \times(x, y)))$.

By our definition of primitive recursion, $\times(x, y) = x \times y$ is primitive recursive. $\square$

Shortcomings of primitive recursion

The primitive recursive functions were supposed to describe the computable functions.

However, in the early 1920s the mathematician Wilhelm Ackermann defined what is now known as the Ackermann function, which is computable but **not** primitive recursive.[1]

[1] `http://gdz.sub.uni-goettingen.de/en/dms/loader/img/?PPN=PPN235181684_0099&DMDID=DMDLOG_0009`

Ackermann function

The Ackermann function is defined as follows:

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)), & m > 0 \text{ and } n > 0. \end{cases}$$

Ackermann function

The Ackermann function is defined as follows:

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)), & m > 0 \text{ and } n > 0. \end{cases}$$

$$A(0,n) = n+1$$
$$A(1,n) = n+2$$
$$A(2.n) = 2n+3$$
$$A(3,n) = 2^{n+3} - 3$$
$$A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3} - 3$$

Ackermann function (cont)

- The Ackermann function is (total) computable, but not primitive recursive.
- It can be shown that the definition of recursion for partial recursive function puts a limit on how "fast" they can grow; the Ackermann function grows faster than any primitive recursive function.

What now?

So we've learned that the primitive recursive functions do not encompass all of the computable functions.

Fortunately, only a small addition is needed to fix this.

Unbounded search

We introduce a sixth type of function, and call the smallest class of functions that satisfy the primitive recursive rules and the following rule the partial recursive functions:

6. Unbounded search (μ-recursion): If $\bar{x}(x_1, \ldots, x_n, \theta(\bar{x}, y))$ is a partial recursive function of $n + 1$ variables, and we define $\psi(\bar{x})$ to be the least y such that $\theta(\bar{x}, y) = 0$ and $\theta(\bar{x}, z)$ is defined for all $z < y$, then ψ is a partial recursive function of n variables.

The notation is dumb, not you

The definition given before is dfficult to parse and phrased unintuitively. Additionally, the precise definition is unimportant for our purposes.So, we summarize it below:

The notation is dumb, not you

The definition given before is dfficult to parse and phrased unintuitively. Additionally, the precise definition is unimportant for our purposes.So, we summarize it below:

6 tells us that, given a partial recursive function, there is a function (μy) (read "the least") that can tell you the least y at which some relation on f holds.

For example, $(\mu y)(y + 5 > 8) = 4)$, while $(\mu y)(y + 5 < 3)$ diverges.

Note that unbounded search allows for partial functions, something which primitive recursion did not.

Coding

Although we've only used functions on the natural numbers so far, we are not quite limited to them.

A coding function is a computale bijection between $\mathbb{N}$ and some set S.
Sets that can be coded into the natural numbers are called effectively countable.

Coding

Although we've only used functions on the natural numbers so far, we are not quite limited to them.

A coding function is a computale bijection between $\mathbb{N}$ and some set S.
Sets that can be coded into the natural numbers are called effectively countable.

Turing machines can compute with coded input by either:

- Decoding the input, and running the computation on that, or
- Computing on the coded input, and return coded output.

Some coding functions

- $\langle x, y \rangle = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$ is known as the pairing function.
- The function τ codes the set of all finite subsets of natural numbers to the natural numbers. It is given by

$$\tau : \bigcup_{k \geq 0} \mathbb{N}^k \to \mathbb{N}$$

$$\tau(\emptyset) = 0,$$

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \cdots + 2^{a_1+\cdots+a_k+k-1}.$$

Coding Turing machines

It turns out that the set of all Turing machines is effectively countable!

In short, we can use the pairing function to code the quadruples, and τ to code the sets of natural numbers, and then "squeeze" the codes together.
It is important to note that this is only one of many bijections between the Turing machines and $\mathbb{N}$.

Coding Turing machines

It turns out that the set of all Turing machines is effectively countable!

In short, we can use the pairing function to code the quadruples, and τ to code the sets of natural numbers, and then "squeeze" the codes together.
It is important to note that this is only one of many bijections between the Turing machines and $\mathbb{N}$.

Not every number coded this way corresponds to a Turing machine; there will be a lot of useless codes.
However, every Turing machine is given by a code.

Coding Turing machines (cont)

We call the method we use to encode the Turing machines an enumeration.

The number assigned to a Turing machine by a fixed enumeration is called the index of that Turing machine.

Padding Lemma

One consequence of enumeration is the Padding Lemma.

As we saw before, there are many indexes which don't encode functional Turing machines.

However, the Padding Lemma states that, given any index of a Turing machine M, there is a larger index which codes a machine that computes the same function as that computed by M.

A universal Turing machine

From the enumeration of the Turing machines and the pairing function, we can define what is known as a universal Turing machine, given by

$$U(\langle e, x\rangle) = \varphi_e(x);$$

where ϕ_e is the eth Turing machine in our enumeration.

A universal Turing machine

From the enumeration of the Turing machines and the pairing function, we can define what is known as a universal Turing machine, given by

$$U(\langle e, x\rangle) = \varphi_e(x);$$

where ϕ_e is the *e*th Turing machine in our enumeration.

This function can replicate every other Turing machine. Again, by the Padding Lemma, there are infinitely many universal Turing machines.

“Wait! you missed something!”

Good question: “But wait! Are the partial recursive functions the computable functions?”

“Wait! you missed something!”

Good question: “But wait! Are the partial recursive functions the computable functions?”

Good answer: “Yes.”

“Wait! you missed something!”

Good question: “But wait! Are the partial recursive functions the computable functions?”

Good answer: “Yes.”

Another good question: “I thought the Turing machines were equivalent to the computable functions?”

“Wait! you missed something!”

Good question: “But wait! Are the partial recursive functions the computable functions?”

Good answer: “Yes.”

Another good question: “I thought the Turing machines were equivalent to the computable functions?”

Another good answer: “That they are.”

Church-Turing Thesis

Alan Turing and his advisor Alonzo Church conjectured that the class of Turing computable functions and the class of partial recursive functions are one and the same.[2]

There are many models ouf computability, in fact:

- Lambda calculus
- Register machines
- Nondeterministic Turing machines

[2] `https://webspace.princeton.edu/users/jedwards/Turing%20Centennial%202012/Mudd%20Archive%20files/12285_AC100_Turing_1938.pdf`

Noncomputable functions

We define the halting function as follows:

$$h(x) = \begin{cases} 1 & \text{if } \varphi_x(x) \text{ halts} \\ 0 & \text{if } \varphi_x(x) \text{ does not halt.} \end{cases}$$

Noncomputable functions

We define the halting function as follows:

$$h(x) = \begin{cases} 1 & \text{if } \varphi_x(x) \text{ halts} \\ 0 & \text{if } \varphi_x(x) \text{ does not halt.} \end{cases}$$

To show that h is not computable, we define the following function:

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{if } h(x) = 1 \\ 1 & \text{if } h(x) = 0. \end{cases}$$

Noncomputable functions (cont)

Notice that if h is computable, then f must be as well. However, for every possible index x,

$$f \neq \varphi_x.$$

So f is not on the list of computable functions, meaning f is not computable. Thus, the halting function h is not computable.

What next?

- Parameterization
- Recursion theorem
- Applications

References

[1] Rebecca Weber, *Computability theory*, Student Mathematical Library, American Mathematical Society, Providence, RI, 1977. MR2920681

Thank you!