CDM

Other Models

Klaus Sutner

Carnegie Mellon University

Alternatives

History shows that there are several suprisingly different ways to define computability:

- J. Herbrand, K. Gödel: systems of equations.
- A. Church: λ-calculus.
- A. Turing: Turing machines.
- Minsky, Shepherdson, Sturgis: register machines.
- S. C. Kleene: μ-recursive functions.
- E. Post: production systems.
- A. A. Markov: Markov algorithms (string rewriting).
- Z. Manna: while programs.

Why Not Just Turing Machines?

Turing machines naturally compute partial functions on words:

$$f\colon \Sigma^\star \nrightarrow \Sigma^\star$$

That is very convenient for complexity theory, but slightly less so for arithmetic functions

$$f\colon \mathbb{N}^k \nrightarrow \mathbb{N}$$

For this to work we need to code natural numbers as strings (e.g. by writing them in binary).

We can avoid coding entirely by using a machine model that manipulates numbers directly: in that sense, register machines are preferable.

The Misery of TMs

Constructing an actual Turing machine for any particular purpose is quite tedious. Try, for example, to construct a TM that performs multiplication when the arguments are given in binary.

Plus, there are endless technical issues: read/write tapes, read-only tapes, write-only tapes, one-way infinite tapes, multiple tracks, ... None of this is hugely complicated, but it requires careful attention to details. Many arguments in complexity theory depend on these gory details.

The Glory of TMs

One of the reason’s Turing’s 1936 paper is so fundamentally important is that he makes a huge effort to convince the reader that “effective calculability” in the intuitives sense corresponds exactly to the power of his machines.

See Turing 36.

Turing’s argument is utterly compelling (and, as far as Gödel was concerned, ended the discussion as to how computability should be defined).

Equations and Computability

Here is a way to tackle the computability question that stays close to standard mathematical practice: write down equations and solve them to obtain arithmetic functions.

We will allow the use of constant 0 and the successor function $x \mapsto x + 1$. Recall that we often write x^+ rather than $x + 1$.

Consider the following equation $\mathcal{E}$:

$$f(x) = x^{++}$$

Intuitively, $\mathcal{E}$ defines the operation "plus two." For example

$$f(2) = 2^{++} = 3^+ = 4$$

This may seem utterly braindead, but hold on.

Numerals

The last chain of equations assumes basic arithmetic, but this is really backwards: we want our equations to define arithmetic, not the other way around.

To fix this problem, we will use numerals denoting natural numbers:

$$\underline{0} \qquad \underline{1} = \underline{0}^+ \qquad \underline{2} = \underline{1}^+ \qquad \underline{3} = \underline{2}^+ \qquad \ldots$$

In general, we have precisely one numeral $\underline{n}$ for every natural number n.

Writing $\underline{2}$ instead of $\underline{0}^{++}$ is just syntactic sugar, nothing new is happening here.

The previous example really should be written

$$f(\underline{2}) = f(\underline{0}^{++}) = \underline{0}^{++\,++} = \underline{1}^{+++} = \underline{2}^{++} = \underline{3}^{+} = \underline{4}$$

WTF?

This is arithmetic; very simple arithmetic, but still:

$$f(2) = 2^{++} = 3^{+} = 4$$

This is just string rewriting, glorified word processing:

$$f(\underline{2}) = f(\underline{0}^{++}) = \underline{0}^{++\,++} = \underline{1}^{+++} = \underline{2}^{++} = \underline{3}^{+} = \underline{4}$$

Needless to say, computers are very good at string rewriting.

Example: Addition

Time for a slightly more ambitious example. As Dedekind pointed out in the late 19th century, we can define addition by the following system of equations $\mathcal{E}$:

$$f(x, \underline{0}) = x$$

$$f(x, y^+) = f(x, y)^+$$

We can check that $\mathcal{E}$ really defines addition:

$$f(\underline{3}, \underline{2}) = f(\underline{3}, \underline{1}^+) = f(\underline{3}, \underline{1})^+ = f(\underline{3}, \underline{0}^+)^+ = f(\underline{3}, \underline{0})^{++} = \underline{3}^{++} = \underline{4}^+ = \underline{5}$$

Equational Reasoning

If we wanted to be strictly formal about this, we would need to spell out a few rules on how to manipulate our equations.

You already know these rules from high school:

- Substitution:
  We can replace a free variable everywhere in an equation by a numeral.

- Replacement:
  A term $f(\underline{a_1}, \ldots, \underline{a_k})$ can be replaced by a numeral $\underline{b}$ if we have already proven $f(\underline{a_1}, \ldots, \underline{a_k}) = \underline{b}$.

$3 + 2 = 5$

(1)	$f(\underline{3}, \underline{0}) = \underline{3}$	subst of $\mathcal{E}_1$
(2)	$f(\underline{3}, \underline{1}) = f(\underline{3}, \underline{0})^+$	subst. $\mathcal{E}_2$
(3)	$f(\underline{3}, \underline{2}) = f(\underline{3}, \underline{1})^+$	subst. $\mathcal{E}_2$
(4)	$f(\underline{3}, \underline{1}) = \underline{3}^+ = \underline{4}$	repl. (1), (2)
(5)	$f(\underline{3}, \underline{2}) = \underline{4}^+ = \underline{5}$	repl. (4), (3)

Tedious and blindingly boring for a human, but completely mechanical and easy to implement in any language with good support for pattern matching.

Computations and Proofs

Note that the last table can be viewed either as

- a detailed description of the computation that shows that $f(3,2) = 5$, or as
- a detailed proof of the assertion that indeed $f(3,2) = 5$.

This is true in general: one can think of computations as proofs and, conversely, of proofs as computations. Both a built according to the same principles.

Willard Van Orman Quine

The utterly pure theory of mathematical proof and the utterly technological theory of machine computation are at bottom one, and the basic insights of each are insights of the other.

W.V.O. Quine, On The Application of Modern Logic

Equational Derivations

This idea extends naturally to arbitrary systems of equations $\mathcal{E}$: we can manipulate the equations to derive assertions about concrete numerals. With a little more effort we could define a notion of derivability (or provability), usually written

$$\mathcal{E} \vdash f(\underline{a_1}, \ldots, \underline{a_k}) = \underline{b}$$

Thus a derivation is a sequence $E_1, \ldots, E_n$ of equations where either $E_k \in \mathcal{E}$ or E_k is obtained by substitution or replacement from an earlier equation in the sequence.

This is entirely similar to Gödel style proofs in first-order logic.

Herbrand-Gödel Computability

Definition

A partial function $\mathbf{f}\colon \mathbb{N}^k \nrightarrow \mathbb{N}$ is Herbrand-Gödel computable if there is a finite system of equations $\mathcal{E}$ that has an k-ary function symbol f such that

$$\mathcal{E} \vdash f(\underline{a_1}, \ldots, \underline{a_k}) = \underline{b} \iff \mathbf{f}(a_1, \ldots, a_k) \simeq b$$

for all $a_i, b \in \mathbb{N}$.

Note the hidden existential quantifier: $\mathbf{f}(\boldsymbol{a}) \simeq b$ iff there exists a corresponding derivation in $\mathcal{E}$ Computing the value of $\mathbf{f}$ is thus a search problem.

Comments

Note that we assume $\mathbf{f}$ only to be a partial function: for some inputs $\underline{\boldsymbol{a}}$ it may not be possible to derive $f(\underline{\boldsymbol{a}}) = \underline{b}$ for any b. This is a bit of a nuisance, but remember the comment at the end of the section on evaluation: divergence is your friend.

Other, traditional names in the literature:

- partial recursive functions or
- general recursive functions.

This is rather misleading, since recursion (or inductive definition) is not really the basic design principle here.

Wisdom

Science advances one funeral at a time.

Max Planck

Warning: Identity

Also note the mysterious notation, sometimes referred to as Kleene equality:

$$\alpha \simeq \beta$$

This means that either

- both α and β are defined (the computations involved all terminate) and have the same value, or
- both α and β are undefined (some computation diverges).

By contrast, we write $\alpha = \beta$ if convergence is not an issue. For example, for primitive recursive functions there is no problem.

In computability theory, identity is more complicated than you might think.

Good News

We have not given a careful and detailed description of what we mean by a "system of equations," but it should be rather clear that any primitive recursive function can be defined by such a system.

- The basic functions successor, zero and projection are all defined in terms of equations.
- Composition and primitive recursion are also defined in terms of equations (rename and combine).

Exercise

Verify the last two claims by writing down the appropriate systems of equations.

More Good News

Recall our definition of the Ackermann function:

$$\begin{aligned} A(0,y) &= y^+ \\ A(x^+,0) &= A(x,1) \\ A(x^+,y^+) &= A(x,A(x^+,y)) \end{aligned}$$

It follows immediately that A is Herbrand-Gödel computable: remember our list-style approach to computing A.

Alas, it is far from clear that the evaluation function from above is similarly Herbrand-Gödel computable (it is, but it is a pain to prove this directly, we'll take a different route).

Bad News

Not every system of equations defines a function. Here are two counterexamples (we'll drop the underscores in numerals from now on).

$$\begin{aligned} f(0,0) &= 0 \\ f(x^+, y) &= f(x, y^+) \end{aligned}$$

$$\begin{aligned} g(x,0) &= x^+ \\ g(x, y^+) &= g(x^+, y) \\ g(x^+, y^+) &= g(x, g(x,y)) \end{aligned}$$

Exercise

Figure out what's wrong with these systems of equations.

Closing the Gap

Systems of equations are naturally very appealing to any math person, but less so from a computer science perspective: we need to filter out the “right” systems of equations.

Unfortunately, as it turns out, there is no algorithm to do this: we can not tell whether a system of equations defines an arithmetic function.

It is therefore rather tempting to try to expand primitive recursive functions in a different way, by adding some other construction method that closes the gap to full computability.

But beware . . .

One needs the constraints imposed by the Herbrand-Gödel approach: otherwise one can use systems of equations to define much more complicated functions.

The problem is that sometimes there is no finite way to deduce the value of a function. For example, consider

$$f(x) = 2 \cdot f(x^+)$$

Clearly, the only solution is $f(x) = 0$, but we cannot tell in finitely many steps: if we only consider arguments $z \leq k$, then

$$f(z) = 2^{k-z}$$

also works.

(General) Recursive Functions

Definition

The clone of μ-recursive functions is defined like the primitive recursive functions, but with one additional operation: unbounded search.

$$f(\boldsymbol{x}) = \min\big(\, z \mid g(z, \boldsymbol{x}) = 0 \,\big)$$

Here g is required to be total.

Thus, $f(\boldsymbol{x}) = 3$ means $g(3, \boldsymbol{x}) = 0$, but $g(2, \boldsymbol{x}), g(1, \boldsymbol{x}), g(0, \boldsymbol{x}) > 0$.

More Convergence

Note f is intuitively computable whenever g is: we just conduct a potentially unbounded search for the least z such that $g(z, \boldsymbol{x}) = 0$.

As a result, even though we insist that g is total, f will in general just be a partial function: for some $\boldsymbol{x}$ there may well be no suitable witness z.

Warning: one cannot allow for g itself to be partial.

Exercise

Figure out why g partial would seem to ruin computability.

Terminology

This last clone of intuitively computable functions is due to Kleene.

The notion μ-recursive function comes from the fact that in the older literature one often finds the more compact and rather cryptic notation

$$f = \mu\, g$$

rather than a reference to $\min$.

So now we have primitive recursive functions, register machine computable functions, Herbrand-Gödel computable (partial recursive) functions, and μ-recursive functions (and, from 251, Turing machine computable functions).

How do these relate?

Comparisons

First off, primitive recursive is weaker than all the others (e.g., Ackermann does not work).

It is a labor of love to show that Turing machines can simulate register machines, and vice versa.

One important point here is that the simulation is effective in the sense that there is a primitive recursive function σ so that the Turing machine with index e is simulated by the register machine with index $\sigma(e)$, and similarly for the opposite direction.

Exercise

Figure out the details of these simulations.

The Rest

So we are left with the question:

- What is the relationship between
  - register machine computable,
  - μ-recursive and
  - Herbrand-Gödel computable?

HG implies Mu

Suppose we have a finite system $\mathcal{E}$ of equations that defines a (partial) function $\mathbf{f}$.

To show that $\mathbf{f}(x) \simeq y$ we need to construct a derivation that uses only the given equations, plus substitution and replacement. With modest effort one can show that this whole machinery is primitive recursive in the following sense: there is a p.r. relation D such that

$$D(t, x, y) \iff t \text{ is a derivation of } f(\underline{x}) = \underline{y} \text{ from } \mathcal{E}$$

But then we can simply perform an unbounded search for the least such t and extract the corresponding y.

Mu implies HG

We need to show how to express the min operator in terms of equations. Here is a trick due to Kleene (1952).

Assume $f(x) = \mu z(g(z,x) = 0)$. Introduce three new function symbols α, β and G with equations

$$\begin{aligned} \alpha(x, y^+) &= x \\ \beta(x, 0) &= x \\ G(0, x) &= g(0, x) \\ G(y^+, x) &= \alpha(g(y^+, x), G(y, x)) \\ f(x) &= \beta(z, G(z, x)) \end{aligned}$$

Ponder deeply. $\square$

Example

Note that $\alpha(x, 0)$ and $\beta(x, y^+)$ are both undefined.

Suppose

$$g(0,5) = 3, \qquad g(1,5) = 7, \qquad g(2,5) = 0$$

We need to derive $f(5) = 2$.

Let's calculate a few values for G:

$$\begin{aligned}
G(0,5) &= g(0,5) = 3 \\
G(1,5) &= \alpha(g(1,5), G(0,5)) = g(1,5) = 7 \\
G(2,5) &= \alpha(g(2,5), G(1,5)) = g(2,5) = 0 \\
G(3,5) &= \alpha(g(3,5), G(2,5)) = \uparrow \\
G(4,5) &= \alpha(g(4,5), G(3,5)) = \uparrow
\end{aligned}$$

Example, Contd.

Now recall $f(5) = \beta(z, G(z, 5))$, which looks bad since there is a free variable on the RHS.

But substituting $z \mapsto 0$ or $z \mapsto 1$ produces a divergent term on the right.

Substituting $z \mapsto 2$ produces $f(5) = \beta(2, G(2, 5)) = 2$.

And substituting $z \mapsto r$ for $r > 2$ also produces a divergent term on the right.

Done.

RM versus Mu

With a bit of effort one can convince oneself that μ-recursive functions can be computed by a register machine: the unbounded search is no problem, we can have a loop that increments a register until a witness is found.

In the opposite direction, each step in a computation of a register machine is primitive recursive, and by adding one unbounded search we can express computations of a register machine in terms of a μ-recursive function.

So, they are all equivalent, and there are simple (primitive recursive) translations between the different models.

Church’s Theory of Function Application

A. Church proposed an exceedingly elegant and highly abstract way to define computable functions: the λ-calculus. In this model, there are only functions; there are no special argument objects to which the functions could be applied, so functions have to be applied to other functions.

This is at odds with one’s intuition that a function should somehow be of higher type than the arguments it is applied to, and takes a bit of getting used to.

The Calculus

We have a rectype of λ terms or combinators that are constructed as follows:

- There is a countable collection of variables x, y, x_i, $\ldots$
- We use special symbols λ, $($, $)$ and . (period)
- All variables are terms, and the constructors are

  Application (MN) is a term for terms M and N.

  Abstraction $(\lambda x.M)$ is a term for x a variable, M a term.

We can define free and bound variables in the obvious manner.

Abbreviations

$$\lambda x_1 \ldots x_k.M \quad \text{for} \quad \lambda x_1(\lambda x_2(\ldots(\lambda x_k.M)\ldots))$$

$$M_1 M_2 \ldots M_k \quad \text{for} \quad (\ldots((M_1 M_2)M_3)\ldots M_k)$$

Thus we implicitely curry: we decompose functions with multiple arguments into a chain of functions (functionals) with single arguments.

This is different from ordinary use, and can be a little confusing: one does typically not think of $f(x,y)$ as $f(x)(y)$.

Also, as usual, one omits parens whenever no harm results.

Reduction Rules

α-Reduction $\lambda x.M \xrightarrow{\alpha} \lambda y.M[x/y]$

Here y must not occur in M.

β-Reduction $(\lambda x.M)N \xrightarrow{\beta} M[x/N]$

Here variables free in N must remain free.

The goal is to apply α and β reductions until no further "simplifications" are possible and we have a term in normal form, an irreducible term.

We can think of the normal form as the "value" of the original term. Applying reductions means we are computing the value of a term.

β-Equivalence

M and N are β-equivalent if M and N can be transformed into the same term by a sequence of α/β reductions.

Notation: $M \overset{\beta}{=} N$.

There are other reductions such as η-reduction:

$$\lambda x.Mx \xrightarrow{\eta} M$$

provided that x is not free in M.

This is a kind of extensionality principle, but we will not pursue this here.

What Could Go Wrong?

Just about everything.

- A term may not have a normal form.
  Example: $D = \lambda x.xx$, then $DD \xrightarrow{\beta} DD$.
- A term might have multiple normal forms.
  Fortunately, Church and Rosser showed that this is not possible.
  The proof is hard.
- Not all attempts at reduction may lead to the normal form, even if it exists.
  Example: $(\lambda xy.y)(DD)z$ fails if one tackles the λ in D.

Note that all these properties fit well if we are trying to use the λ-calculus to express computability.

Fixed Points Everywhere

Theorem (Kleene, Turing, Curry)

There is a fixed-point operator $\mathcal{Y}$ such that, for every term M:
$\mathcal{Y}M \overset{\beta}{=} M(\mathcal{Y}M)$.

Proof.

Let $D = \lambda x.M(xx)$ so that $Dt \overset{\beta}{=} M(tt)$ for any term t.

Hence $DD \overset{\beta}{=} M(DD)$ and we have a fixed point.

We can abstract M from this and get

$$\mathcal{Y} = \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

□

Numerals

One might suspect that we need to somehow add natural numbers to the calculus if we want to represent arithmetic functions.

Fortunately, they are already there, albeit in a somewhat opaque manner: we can exploit iteration to express naturals:

$$f^n(x) \quad \text{somehow represents } n$$

More precisely, we have (Church) numerals

$$\underline{n} = \lambda f x. f^n(x)$$

Note that f is just syntactic sugar, we don't have special variables for functions.

Arithmetic

Successor	$\lambda nfx.f(nfx)$
Addition	$\lambda mnfx.mf(nfx)$
Multiplication	$\lambda mnf.m(nf)$
Exponentiation	$\lambda be.eb$
Predecessor	$\lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$

Some basic arithmetic functions expressed as λ-terms.

Exercise

Verify that these definitions work as advertised.

Logic

True	$\lambda xy.x$
False	$\lambda xy.y$
And	$\lambda pq.pqp$
Or	$\lambda pq.ppq$
Not	$\lambda p.p$ True False
ITE	$\lambda pqr.pqr$

Some basic propositional logic expressed as λ-terms.

Exercise

Verify that these definitions work as advertised.

Combinatorics

Pair	$\lambda xyf.fxy$
First	$\lambda p.p$ True
Second	$\lambda p.p$ False
Nil	$\lambda x.$True

Foundations for lists as λ-terms.

Exercise

Verify that these definitions work as advertised. Devise a test for a list being empty.

λ-Definability

We can now say that $f\colon \mathbb{N}^k \nrightarrow \mathbb{N}$ is λ-definable if there is a term M such that $f(a_1, \ldots, a_k) \simeq b$ iff $M \underline{a}_1 \underline{a}_2 \ldots \underline{a}_k$ reduces to $\underline{b}$.

There are other ways to set up numerals, but one can show that they all lead to the same fundamental theorem.

Theorem (Church, Rosser, Kleene)

An arithmetic function is computable if, and only if, it is λ-definable.

Proofsketch

It is a labor of love to express primitive recursive functions as λ-terms. For example,

$$\lambda x.\underline{0},\ \lambda zfx.f(zfx),\ \lambda x_1 \dots x_k.x_i$$

represent the atomic functions, pairing can be handled via $\lambda yzfgx.yf(zgx)$, and so forth.

The hard part is to deal with unbounded search as in $f(\boldsymbol{x}) \simeq \min\big(z \mid g(\boldsymbol{x}, z) = 0 \big)$. But then $f(\boldsymbol{x}) = F(\boldsymbol{x}, 0)$ where

$$F(\boldsymbol{x}, z) = \begin{cases} z & \text{if } g(\boldsymbol{x}, z) = 0, \\ F(\boldsymbol{x}, z+1) & \text{otherwise.} \end{cases}$$

But then we can exploit the fixed-point theorem to show that F is λ-definable.

Opposite Direction

To show that λ-definable implies computable, first note that the predicate "t codes a β-reduction from M to N" is primitive recursive: we can arithmetize everything to translate all objects to natural numbers.

Each step in a β-reduction is clearly primitive recursive, as is the whole sequence.

But then we need to add just one unbounded search to find the appropriate t.

The Key Ideas

We can specify a model $\mathfrak{M}$ of computation by defining

- a space $\mathcal{C}$ of possible configurations (snapshots),
- a “one-step” relation,
- an input and output convention,
- a coding convention (if needed).

The details vary greatly, but we always have the same pattern.

Major Warning: Minute details about input/output/coding conventions become really important in low complexity classes; higher up they are mostly interchangeable.

Whole Computation

Given a one-step relation $C \mathrel{\vert\!\frac{1}{\mathfrak{M}}} C'$, multiple steps and whole computations are defined in the obvious way:

$$C \mathrel{\vert\!\frac{0}{\mathfrak{M}}} C' :\Leftrightarrow C = C'$$

$$C \mathrel{\vert\!\frac{t}{\mathfrak{M}}} C' :\Leftrightarrow \exists\, C'' \;\; C \mathrel{\vert\!\frac{t-1}{\mathfrak{M}}} C'' \wedge C'' \mathrel{\vert\!\frac{1}{\mathfrak{M}}} C'$$

$$C \mathrel{\vert\!\frac{}{\mathfrak{M}}} C' :\Leftrightarrow \exists\, t \;\; C \mathrel{\vert\!\frac{t}{\mathfrak{M}}} C'$$

A computation (or a run) of $\mathfrak{M}$ is a sequence of configurations C_0, C_1, C_2, . . . where $C_i \mathrel{\vert\!\frac{}{\mathfrak{M}}} C_{i+1}$.

Input/Output

One needs a way to provide input from some set X, a map

$$\mathsf{inp} : X \to \mathcal{C}$$

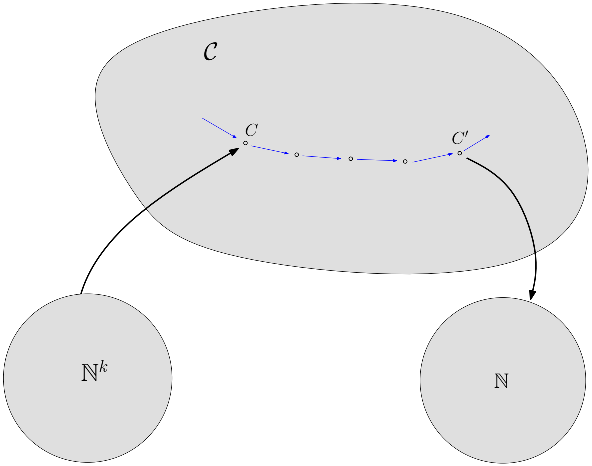
as well as an output map to some set Y:

$$\mathsf{outp} \colon \mathcal{C} \nrightarrow Y$$

Both maps are very typically very simple, essentially just a bit of re-formatting (they do not contribute to the complexity of the computation).

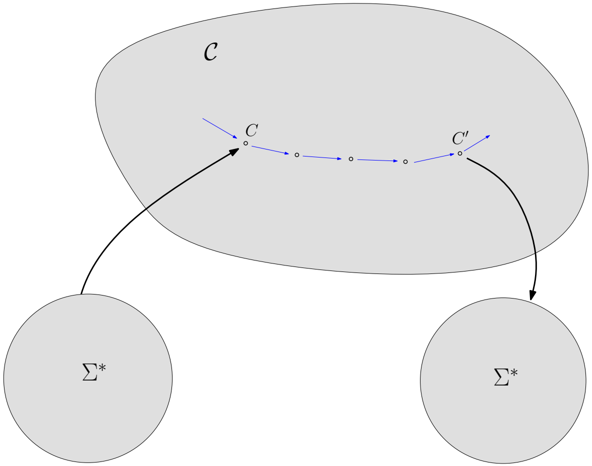
A minor technical issue: the output map may only be defined on some of the configurations (the “halting” configurations).

Models of Computation



The number-theoretic scenario: input and output are natural numbers.

Models of Computation, II



The string scenario: input and output are words over some alphabet.

Empirical Observation

For all the models mentioned so far, one can simulate model $\mathfrak{M}$ in any other model $\mathfrak{M}'$ (excluding, of course, the primitive recursive ones).

In fact, the cross-model simulation maps are all very simple (say, primitive recursive).

Of course, the length of a computation may differ a bit in different models, a major issue in complexity theory.

Church's Proposals

- Church's Thesis I (1934):
  Effectively calculable iff λ-definable.
- Church's Thesis II (1935):
  Effectively calculable iff general recursive.
- Gödel rejected these early proposals by Church.

Gödel's Position

But then in 1936, A. Turing introduced his theory of computability based on Turing machines. Gödel responded most enthusiastically to Turing's work:

> *This concept, . . . is equivalent to the concept of a "computable function of integers" . . . The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.*

Gödel Later

But I was completely convinced only by Turing's paper.

See A. Turing [1936] and the almost simultaneous paper by E.L. Post [1936]. As for previous equivalent definitions of computability, which, however, are much less suitable for our purpose, see A. Church [1936].

Gödel explains his position in 1968.

He always gave full credit to Turing, never to Church or himself.

Church's Response to Turing

The first has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately, i.e., without the necessity of proving preliminary theorems.

Church, in his 1937 review of Turing [1936].

Church-Turing Thesis

Church was the first to become convinced that

> All (reasonable) models of computation are equivalent, and correspond exactly to our intuitive notion of computability.

So far (Fall 2018), there are no interesting objections to this idea. So from now on we will often simply talk about “computable” functions and reference specific models only when necessary.

The Cult of Hypercomputation

A lot of people would agree that the Church-Turing thesis also has a physical interpretation:

All (reasonable) models of computation correspond exactly to physically realizable computability.

This is much more problematic, since our actual universe appears to be finite in many ways–we would need to ignore these constraints.

Alas, we have nothing resembling an axiomatization of physics, so there is the logical possibility that some currently unknown physical phenomenon would allow one "to break through the Turing limit."

Possible, but not very likely. Forcryingoutloud, we can't even factor numbers.