# Logic, Methodology and Philosophy of Science

Proceedings of the 14th International Congress (Nancy)

## Logic and Science Facing the New Technologies

Edited by

Peter Schroeder-Heister,

Gerhard Heinzmann,

Wilfrid Hodges,

and

Pierre Edouard Bour

ISBN 978-1-84890-169-8

College Publications
Scientific Director: Dov Gabbay
Managing Director: Jane Spurr

http://www.collegepublications.co.uk

Printed by Lightning Source, Milton Keynes, UK

---

# On the Church-Turing Thesis and Relative Recursion

Yiannis N. Moschovakis

## 1 Introduction

The *Church-Turing Thesis* is the claim that for every function $f : \mathbb{N}^n \to \mathbb{N}$ on the natural numbers $\mathbb{N} = \{0, 1, \ldots\}$,

CT: *$f$ is computable* $\iff$ *$f$ can be computed by a Turing machine.*

It implies that for every relation $R$ on the natural numbers or (via an effective coding) on the set $\Lambda^*$ of *strings* from some finite alphabet $\Lambda$,

*$R$ is decidable* $\iff$ *its characteristic function is Turing computable.*

CT was postulated (in different but equivalent forms) by (Church 1935; 1936*b*;*a*) and (Turing 1936), who applied it immediately to prove the *undecidability of provability in first order logic*. This solved the classical *Entscheidungsproblem*—or showed that it was "unsolvable", depending on your point of view; and similar invocations of CT have been used to establish some of the most important applications of logic to mathematics and computer science in the 20th century, including the unsolvability of Hilbert's 10th problem by Matiyasevich (after Davis, Putnam and Robinson), the construction of finitely generated, finitely presented groups whose word problem is unsolvable (Novikov and Boone), etc.

There was some initial scepticism (cf. (Moschovakis 1968)), but there is no doubt that the Church-Turing Thesis is almost universally accepted today, so much so that it is usually invoked with no explicit mention. At the same time, foundational questions about its epistemological status and *what exactly it means* continue to generate considerable discussion: is it a "log-

P. Schroeder-Heister, G. Heinzmann, W. Hodges, and P. E. Bour (eds.), *Logic, Methodology and Philosophy of Science. Proceedings of the Fourteenth International Congress (Nancy), 179–200. 2014.*

ical", a "mathematical" or an "empirical truth"? And more significantly: can it be *proved* from simple, plausible assumptions?[1]

It is not my purpose here to give another proof of CT or a critical analysis of the arguments that have been given for it—a daunting task, given the vast material published on the problem. My aim is to introduce and discuss the *Thesis on Relative Recursion* RRT, a principle which is related to but very different from CT, not only in content but in kind; and to show that CT can be reduced to the conjunction of RRT and a nearly universally accepted view of *what the natural numbers are*. The move shifts the discussion about the meaning and truth of CT to the corresponding questions about RRT, which (I think) are substantially simpler.

The brief, last paragraph 8.1 summarizes what I believe is achieved in this article.

### 1.1 About algorithms

It is often assumed that CT is equivalent to

$$\mathsf{CT}^*: \quad f:\mathbb{N}^n\to\mathbb{N} \text{ } \textit{is computable by an algorithm} \iff f \textit{ can be computed by a Turing machine.}$$

This is, in fact, how (Church 1936*b*) formulates CT, although the earlier (Church 1935) and (Turing 1936) do not mention algorithms.

We have direct intuitions about *algorithms* which can be brought to bear in discussing CT, as it is natural to assume that

(1) *if some algorithm computes a function $f$, then $f$ is computable.*

However, it can be (and has) been argued that we also have independent, direct intuitions about functions on the natural numbers which are *computable by finite means* in Turing's words. In other words, it may well be that CT and $\mathsf{CT}^*$ are both true but they do not have the same meaning, and so arguments in favor of one of them do not necessarily apply to the other. For example: arguments for CT often depend on assumptions about the (natural) *primitives of computation*, while arguments for $\mathsf{CT}^*$ are naturally grounded on explications of *what algorithms are*, which is a difficult (and controversial) subject.

I take (1) to be obviously true, if it is understood correctly, and so algorithms will come up naturally and often in the sequel. However: I take CT to be the "official" version of the Church-Turing Thesis, and I will refrain

[1] Cf. (Gandy 1980; 1995), (Sieg 2002), (Kripke 2000) (which is a recording), (Dershowitz & Gurevich 2008) and the large bibliographies in these papers.

from discussing in any serious way the difficult problem of explicating the notion of algorithm; this is not what this article is about.[2]

## 2 Preliminary remarks,

on three issues which bear on our understanding of the Church-Turing Thesis.

### 2.1 The primitives of computation

(Turing 1936) starts with

> the "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means,

and then he argues (mostly) in the last Section 9 that his "computing machines" capture the natural notion of "calculability". His reasoning is driven by the following statement in the first paragraph of Section 9:

> The real question at issue is "What are the possible processes which can be carried out in computing a [real] number?"

The elementary operations that Turing machines can do involve manipulating tapes with symbols on them, and they are most directly understood as operations on (finite) *strings of symbols*. Turing argues that all *immediate* string operations (which intuitively can be effected in one step) can be simulated in finitely many steps by those basic, Turing machine operations. His arguments "are bound to be, fundamentally, appeals to intuition", he says, "and for this reason rather unsatisfactory mathematically". They are, however, very persuasive and were pivotal in securing the quick acceptance of CT. Gandy's seminal 1980 calls Turing's analysis an "outline of a proof" of[3]

[2] I think that the problem of giving a rigorous, mathematical definition of *algorithms* is very important for the foundations of the theory of computation and has not received the attention it deserves. For my own ideas about it, see (Moschovakis 1998), and for alternative proposals cf. (Gurevich 2000), (Dershowitz & Gurevich 2008) and (Tucker & Zucker 2000).

[3] (Gandy 1980) gives an alternative understanding of CT which limits computability by arbitrary *mechanical devices*, and then gives an explication of what these are and bases on it a proof of CT. Physics enters the picture and CT becomes an *empirical proposition*, burdened by the usual problems: what if, in some distant future, someone builds a *Higgs boson machine* which can use God as an oracle and get answers to arbitrary mathematical questions, perhaps by doing subtle experiments? (Kripke 2000) argues (in a more serious vein) that the empirical understanding of CT is problematic and impossible to settle without a great deal more knowledge about physical laws than physicists have today. He concludes that CT is most coherently understood as a *mathematical proposition* and

> Theorem T. What can be calculated by an abstract human being working in a routine way is [Turing] computable.

### 2.2 Symbolic computation

Built into this picture of the "mindless clerk" scribbling away is the principle that *all computation is symbolic*, which is why the *primitives of computation* are assumed to be operations on strings of symbols. This is a plausible (and popular) slogan which, however, merits some discussion. We will return to it further down.

### 2.3 Input and output

(Church 1936*b*) formulates CT in the form $\mathsf{CT}^*$ above, at least by the words he uses:

> ... every function, an algorithm for the calculation of the values of which exists, is effectively calculable [$\sim$ Turing computable].

He then goes on to explain that for a function $F(n)$ of one positive integer,

> an algorithm consists in a method by which, given any positive integer $n$, a sequence of expressions (in some notation) $E_{n1}, E_{n2}, \ldots, E_{nr_n}$ can be obtained; ... [and in the end] the fact that the algorithm has terminated becomes effectively known and the value of $F(n)$ is effectively calculable. ... If this interpretation or some similar one is not allowed, it is difficult to see how the notion of an algorithm can be given any exact meaning at all.

So Church's rather restricted understanding of "algorithms" invokes again this "all computation is symbolic" principle, albeit somewhat more loosely than Turing's. But I want to stop here on this innocent

> given any positive integer $n$ ...

Exactly how is a positive integer $n$ *given*? Perhaps Church has in mind the rather complex *numeral* for $n$ of the untyped $\lambda$-calculus or, more likely, the term $S^n(0)$ in Herbrand-Gödel-Kleene systems of equations, the unary representation of $n$. But why not use binary notation, as is routinely done today? Or, for that matter, why not "give $n$" to the algorithm by coding the value $F(n)$ in the syntactic expression $E_{n1}$, rendering all further computation redundant?

---

outlines a proof of it, elaborating on arguments which are (at least implicit) in (Turing 1936) and (Church 1936*b*). I will not go into this reasoning here, but I also understand CT as a mathematical proposition on more basic grounds: *it refers essentially to the natural numbers*, and so its truth or falsity depends on what they are.

The joke is old and worn out, but it makes the point: if we are to understand all computation as symbolic, then in addition to identifying the *string primitives* which our (human or mechanical) computing machine can call directly, we must also specify an *input function* which turns a "given $n$" into a string and also an *output function* which decodes the required value from the contents of the tape when the machine stops; and to prove CT, we must argue that these input and output functions are "effective"—in fact "immediately effective"—without benefit of the Church-Turing Thesis.[4]

## 3 Mathematical algorithms

To understand better the connection between algorithms and computability expressed by (1), we discuss briefly two well known, classical algorithms and a simple recursive process which is a variation on many popular themes.

### 3.1 The Euclidean algorithm (before 300 BC)

For $a, b \in \mathbb{N} = \{0, 1, \ldots\}$, $a, b \neq 0$,

$$\gcd(a, b) = \text{the largest number which divides both } a \text{ and } b.$$

It is easy to check that if for $a, b \neq 0$, $\mathrm{rem}(a, b)$ is the *remainder* of $a$ by $b$, the unique number $r$ such that for some $q \in \mathbb{N}$

$$(2) \quad a = qb + r \text{ and } 0 \leq r < b,$$

then

$$(3) \quad \gcd(a, b) = \text{if } (\mathrm{rem}(a, b) = 0) \text{ then } b \text{ else } \gcd(b, \mathrm{rem}(a, b)).$$

This is the basic mathematical fact about the *greatest common divisor* function, it defines it implicitly, and it expresses a *recursive algorithm* for computing it using iterated division:[5]

[4] (Turing 1936) avoids these problems by using machines with no input which compute (the decimal expansions of) real numbers and reading the (infinite) output from the digits "emitted" during the computation; but he would surely need to face up to them to "investigate computable functions of an integral variable" in much the same way, as he says he can do.

[5] Euclid defines first the so-called *subtractive Euclidean algorithm* which uses *anthypheresis*

$$A(x, y) = (\max(x, y) - \min(x, y), \min(x, y)),$$

an operation on unordered pairs of numbers which was very important in Greek mathematics. The relevant recursive equation now is

$$\gcd(x, y) = \text{if } (x = y) \text{ then } x \text{ else } \gcd(A(x, y)),$$

if $\mathrm{rem}(a, b) = 0$, give output $b$,
otherwise replace $(a, b)$ by $(b, \mathrm{rem}(a, b))$ and repeat.

The important mathematical facts about the Euclidean algorithm are the following:

(a) *Correctness*: for all non-zero $a, b \in \mathbb{N}$, the Euclidean terminates and yields $\gcd(a, b)$.

(b) *Primitives*: The Euclidean is an algorithm on $\mathbb{N}$ *from* rem and $=_0$ (equality with 0).

(c) *Complexity*: if $\mathrm{calls}(a, b)$ is the number of *calls to* rem that the Euclidean makes to compute $\gcd(a, b)$, then

$$\mathrm{calls}(a, b) \leq 2 \log_2(b) \quad (\text{for } a \geq b \geq 2).$$

We might be tempted to define $\mathrm{calls}(a, b)$ as the *number of divisions* the algorithm makes to compute $\gcd(a, b)$, the basic *division algorithm* being the most obvious way to get at the remainder. But there is nothing in the specification of the Euclidean (by the recursive equation (3)) which tells us how $\mathrm{rem}(x, y)$ must be obtained whenever it is needed: there might, in fact, be some *fast division algorithm* which is more efficient than the usual division process (as *fast multiplication* is more efficient than the elementary school algorithm for multiplication), or even some very clever, still unknown method which gets $\mathrm{rem}(x, y)$ very quickly without also computing the quotient of $x$ by $y$. None of that matters to the Euclidean which simply needs $\mathrm{rem}(x, y)$ for various pairs $x, y$ in the process of computing $\gcd(a, b)$; this is why we say that the Euclidean is an algorithm *from* rem *and* $=_0$ rather than "from division and $=_0$".

There is a large number of extensions and generalizations of the Euclidean algorithm, from which we mention here just two, also known to the Greeks before Euclid's time.

**The Euclidean on positive real inputs**

The division equation (2) holds for positive real numbers $a, b \in \mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$ and determines a unique *integer quotient* $q = \mathrm{iq}(a, b) \in \mathbb{N}$ and remainder $\mathrm{rem}(a, b) \in \mathbb{R}$. It follows that the basic recursive equation (3) makes sense when $a, b \in \mathbb{R}^+$ and expresses (as before) an algorithm on pairs

and so the subtractive Euclidean is an algorithm from $=$ and $A$. Later on he switches to our version of the Euclidean without much comment, most likely thinking that the subtractive version simply "implements" division by "iterated subtraction".

Perhaps more importantly, Euclid defines his algorithms using *iteration* (like modern *while programs*) rather than recursive equations. The relation between *iterative* and *recursive algorithms* is important but subtle and I will not discuss it in this article, cf. (Moschovakis 1998).

from $\mathbb{R}^+$, except that in this case the computation may go on forever, so that this algorithm computes a *partial function*[6] on $\mathbb{R}^+ \times \mathbb{R}^+$ with values in $\mathbb{N}$; moreover

(4) the Euclidean terminates on $a, b \in \mathbb{R}^+$

$$\iff a \text{ and } b \text{ are commensurable, i.e., } \frac{a}{b} \text{ is a rational number,}$$

an important fact known to the Greeks and their basic method for proving *non-commensurability*.

**The continuous fraction algorithm**

The output of the Euclidean on positive reals, when it converges, is not especially interesting. More basic is the variation of the Euclidean which applies (3) again on pairs of positive real numbers but outputs the (finite or infinite) sequence $q_0, q_1, \ldots$ of the quotients produced during the computation, i.e., the *continuous fraction representation* of the quotient $\frac{x}{y}$. This is an algorithm on $\mathbb{R}^+$ from $\mathrm{rem}, =_0, \mathrm{iq}$ on $\mathbb{R}^+$ and $0, S, \mathrm{Pd}, =_0$ on $\mathbb{N}$, another of the fundamental algorithms of Greek mathematics with important applications in number theory.

### 3.2 The Sturm algorithm (1829)

This computes *the number of real roots* of a polynomial

(5) $p(x) = a_0 + a_1 x + \cdots + a_n x^n$

of degree $\leq n$ with real coefficients in a real interval $(b, c)$. It operates on tuples $(a_0, \ldots, a_n, b, c)$ of real numbers and its primitives are the field operations $0, 1, +, -, \cdot, \div$ and the ordering $\leq$ of $\mathbb{R}$. A simple elaboration of it decides the relation

$$R(a_0, \ldots, a_n) \iff (\exists x \in \mathbb{R})[a_0 + a_1 x + \cdots + a_n x^n = 0].$$

The main subroutine of the Sturm algorithm is a version of the Euclidean, applied to the space of real polynomials of degree $\leq n$ and with a (critical) twist, in which the remainder $r(x)$ at each step is replaced by $-r(x)$. It is an important algorithm and the main algebraic fact extended and used by (Tarski 1951) in his famous proof of the *decidability of the first order theory of* $\mathbb{R}$ as an ordered field.

[6] A partial function $f : X \rightharpoonup Y$ is an ordinary function $f : X_0 \to Y$ on some arbitrary $X_0 \subseteq X$, the *domain of convergence* of $f$. As usual, $f(x)\downarrow \iff x \in X_0$, $f(x)\uparrow \iff x \notin X_0$, and for $f_1, f_2 : X \rightharpoonup Y$,

$$f_1(x) = f_2(x) \iff [f_1(x)\uparrow \ \& \ f_2(x)\uparrow] \vee f_1(x) = f_2(x).$$

### 3.3 The color of leaves

A (finite, rooted, binary, colored) *tree* is a tuple

$$\mathbf{T} = (T, \text{root}, l, r, \text{Leaf}, \text{Red}),$$

where $T$ is a finite set; root $\in T$; Leaf and Red are unary relations on $T$; and

$$l, r : T \setminus \{x \in T \mid \text{Leaf}(x)\} \rightarrowtail T \setminus \{\text{root}\}$$

are injections with disjoint images whose union exhausts $T \setminus \{\text{root}\}$. A *path from* $x_0$ *to* $x_n$ in $\mathbf{T}$ is any sequence $(x_0, \ldots, x_n)$ such that for each $i < n$, $x_{i+1} \in \{l(x_i), r(x_i)\}$. It follows easily that every *node* $x \in T$ is the endpoint of a unique path from root, and that every *maximal path* ends at a leaf. Set[7]

$$\begin{aligned}(6)\quad R(x) &\iff \text{every leaf below } x \text{ is red}\\ &\iff (\text{for every path } (x, x_1, \ldots, x_n))[\text{Leaf}(x_n) \implies \text{Red}(x_n)].\end{aligned}$$

The basic mathematical fact about this relation is the equivalence

$$(7)\quad R(x) \iff \text{if Leaf}(x) \text{ then Red}(x) \text{ else } [R(l(x)\ \&\ R(r(x))];$$

and as with the Euclidean, it expresses a recursive algorithm for deciding $R(x)$:

> if Leaf$(x)$, output the truth value of Red$(x)$,
> otherwise decide $R(l(x))$ and $R(r(x))$ using the same procedure
> and output the Boolean product $R(l(x))\ \&\ R(r(x))$.

This is an abstract version of many standard, recursive (*divide-and-conquer*) algorithms, including the *merge-sort*.[8] It operates on the set $T$ and its primitives are those of the structure $\mathbf{T}$, i.e., root, $l, r$, Leaf and Red.

[7] This is a simplified version of the example in the basic article (Tiuryn 1989), which separates non-deterministic from deterministic recursive computability and also full recursion from *tail recursion*, i.e., iteration.

[8] The merge-sort orders (alphabetizes, sorts) finite sequences from a set $L$ with respect to a given ordering of $L$, and is asymptotically optimal for the number of comparisons it needs to do the job. Its optimality among sorting algorithms (of the appropriate kind) is probably the only lower bound result proved in every introductory course in Computer Science, and so a discussion of it can be found in any standard, introductory text. See (Moschovakis 1998) for a discussion of its significance for the foundations of the theory of algorithms.

This *leaf-color algorithm* can also be applied when $T$ is infinite. In this case, the computation described terminates only on *the well founded part of* $T$

$$\mathrm{WF}(\mathbf{T}) = \{x \in T \mid (\exists n)[\text{every path from } x \text{ has length } \leq n]\}$$

and decides correctly the relation $R$ on $\mathrm{WF}(\mathbf{T})$.[9]

In a second variation we consider *arbitrary, well founded binary trees*, i.e., structures of the form

$$\mathbf{T} = (T, \mathrm{Roots}, l, r, \mathrm{Leaf}, \mathrm{Red})$$

where Leaf and Red are as above; $\mathrm{Roots} \subseteq T$ is a non-empty set of *roots*;

$$l, r : T \setminus \{x \in T \mid \mathrm{Leaf}(x)\} \rightarrowtail T \setminus \mathrm{Roots}$$

are injections with disjoint images; every $x \in T$ occurs on some path which starts with a root; and there are no infinite paths. Now $\mathrm{WF}(\mathbf{T}) = T$ and the algorithm terminates for every $x$ and decides whether all the (finitely many) leaves below $x$ are red.

## 4 Computing on an arbitrary set from specified primitives

The algorithms in Section 3 are very different from those envisioned by Church or expressed by Turing machines. Some of their important features are:

(I) *Arbitrary universe*: They operate on sets other than $\mathbb{N}$ or the strings from some finite alphabet: the real numbers for the variations of the Euclidean and the Sturm and an arbitrary finite or infinite set $T$ for the leaf-color algorithm and its variations.

In particular, the computations defined by them are not "symbolic".

(II) *No input function*: They operate *directly* on their arguments, i.e., there is no intermediary of an input or an output function.

(III) *Use of arbitrary primitives*: They can use (call) specified primitives (constants, functions and relations) on their domain of application which need not be (and often are not) intuitively effectively computable. For example, the Sturm uses the inequality relation on $\mathbb{R}$ which is not decidable in any meaningful sense, and the second variation of the leaf-color algorithm operates on an arbitrary set $T$, for which it does not make sense to ask whether the primitives $\mathrm{Roots}, l.r. \ldots$ are effective—they are just *given*.

[9] By König's Lemma, $x \in \mathrm{WF}(\mathbf{T})$ exactly when no *infinite path* starts from $x$.

One might argue that the specifications we gave in (3) and (7) are not precise or at least not complete, and they do not meet today's standard of rigor unless they are complemented by directions for how to "implement" them. This is one of many legitimate issues which make the foundational problem of explicating the notion of algorithm subtle (and even controversial). But this is not our issue here: what we will do in the next section is to extract from the robust intuitions behind these classical algorithms a precise notion of *computability from arbitrary primitives* for functions and relations on arbitrary sets. This is a natural and useful notion, and we will show in Section 8 that it is closely—and usefully—related to the kind of computability on the natural numbers that the Church-Turing Thesis is about.

## 5 Recursion in an arbitrary partial structure

We outline here very briefly the basic definitions of recursion in first order structures, for the sake of completeness. A full exposition of the (much richer) *recursion in a many-sorted, functional structure* is given in (Moschovakis 1989), and a summary of a mildly restricted case is included in (van den Dries & Moschovakis 2004).

It is convenient to think of a relation $R \subseteq A^n$ as a function $R : A^n \to \{\mathrm{T}, \mathrm{F}\}$ and of an element $c \in A$ as a nullary function with value $c$. This makes it natural to also allow *partial relations* $R : A^n \rightharpoonup \{\mathrm{T}, \mathrm{F}\}$, and so to deal uniformly with (partial) functions, relations and objects.

With these conventions, a *vocabulary* (signature) is a tuple

$$\Phi = (\phi_0, \ldots, \phi_k)$$

of function symbols, together with two functions, arity and sort, which assign to each $\phi_i$ its *arity*, the number of arguments that it expects, and its *sort*, the kind of values it takes, `ind` or `boole`; and a (partial) $\Phi$-*structure* is a tuple

$$\mathcal{A} = (A, \boldsymbol{\Phi}) = (A, \phi_0^{\mathcal{A}}, \ldots, \phi_k^{\mathcal{A}})$$

where each $\phi_i^{\mathcal{A}}$ is a *partial* constant, relation or function on the universe $A$ of the appropriate arity and sort, e.g., if $\mathrm{arity}(\phi_i) = n$ and $\mathrm{sort}(\phi_i) = \mathtt{ind}$, then

$$\phi_i^{\mathcal{A}} : A^n \rightharpoonup A.$$

The $\mathcal{A}$-*terms* (with parameters and conditionals) are defined by the recursion

$$E :\equiv \mathrm{T} \mid \mathrm{F} \mid x \mid \mathsf{v}_j \mid \phi_i(E_1, \ldots, E_n) \mid \text{if } E_0 \text{ then } E_1 \text{ else } E_2$$

where $x$ is any member of $A$; $\{\mathsf{v}_0, \mathsf{v}_1, \ldots\}$ is a fixed sequence of variables of sort `ind`; arity$(\phi_i) = n$ and $E_1, \ldots, E_n$ are of sort `ind`; and in the conditional, $E_0$ is of sort `boole` and sort$(E_1)$ = sort$(E_2)$. The definition also assigns to each term its *parameters* (the members of $A$ which occur in it), its *variables*, and in the obvious way, its *sort*. The sort of the conditional construct is sort$(E_1)$ (= sort$(E_2)$).

A term is *closed* if it has no variables and a *pure* $\Phi$*-term* if it has no parameters. We use the customary notation for *substitution*: if $E(\mathsf{u}_1, \ldots, \mathsf{u}_n)$ is a term in which the distinct variables $\mathsf{u}_1, \ldots, \mathsf{u}_n$ may occur and $u_1, \ldots, u_n \in A$, then $E(u_1, \ldots, u_n)$ is the result of replacing in $E$ each $\mathsf{u}_i$ by $u_i$.

The denotations of closed $\mathcal{A}$-terms are defined as one might expect:

$$\begin{aligned}
&\mathrm{den}(\mathrm{T}) = \mathrm{T}, \quad \mathrm{den}(\mathrm{F}) = \mathrm{F}, \quad \mathrm{den}(x) = x,\\
&\mathrm{den}(\phi_i(E_1, \ldots, E_n)) = \phi_i^{\mathcal{A}}(\mathrm{den}(E_1), \ldots, \mathrm{den}(E_n)),\\
&\mathrm{den}(\text{if } E_0 \text{ then } E_1 \text{ else } E_2) = \text{if } \mathrm{den}(E_0) \text{ then } \mathrm{den}(E_1) \text{ else } \mathrm{den}(E_2).
\end{aligned}$$

We write $\mathrm{den}(\mathcal{A}, E)$ if it is important to specify the structure in which the denotation is computed, and we note that $\mathrm{den}(\mathcal{A}, E)$ need not always converge, because we have allowed partial functions in $\mathbf{\Phi}$.

**Recursive (McCarthy) programs**

An $n$-ary (deterministic) *recursive* $\Phi$*-program*[10] $E$ is a syntactic expression

$$(8) \quad E \equiv E_0(\vec{\mathsf{x}}, \vec{\mathsf{p}}) \text{ where } \{\mathsf{p}_1(\vec{\mathsf{u}}_1) = E_1(\vec{\mathsf{u}}_1, \vec{\mathsf{p}}), \ldots, \mathsf{p}_k(\vec{\mathsf{u}}_k) = E_k(\vec{\mathsf{u}}_k, \vec{\mathsf{p}})\}$$

where the following conditions hold:

(RP1) $\vec{\mathsf{p}} \equiv \mathsf{p}_1, \ldots, \mathsf{p}_k$ is a sequence of distinct function and relation symbols which do not occur in $\Phi$. These are the *recursive variables* of $E$.

(RP2) For $i = 0, \ldots, k$, the *part* $E_i(\vec{\mathsf{u}}_i, \vec{\mathsf{p}})$ of $E$ is a pure term (no parameters) in the vocabulary $\Phi \cup \{\mathsf{p}_1, \ldots, \mathsf{p}_k\}$ whose variables are in the list $\vec{\mathsf{u}}_i$ (where by convention, $\vec{\mathsf{u}}_0 \equiv \vec{\mathsf{x}} \equiv \mathsf{x}_1, \ldots, \mathsf{x}_n$).

(RP3) For $i = 1, \ldots, k$, sort$(E_i)$ = sort$(\mathsf{p}_i)$.

[10] Deterministic recursive programs were introduced by (McCarthy 1963), who used them to develop clean foundations for call-by-value computability from arbitrary, specified primitives. Especially significant was McCarthy's explicit identification of the *conditional* (branching) as an essential ingredient of computation: he used it to give an elegant characterization of the general recursive functions on $\mathbb{N}$ which avoids the non-determinism inherent in the *Herbrand-Gödel-Kleene* systems of (Kleene 1952).

The sort of $E$ is the sort of its *head term* $E_0(\vec{\mathsf{x}}, \vec{\mathsf{p}})$; the *free occurrences of variables* of $E$ are the occurrences of $\mathsf{x}_1, \ldots, \mathsf{x}_n$ in its head; and its *bound occurrences* of variables are those in the lists $\vec{\mathsf{u}}_i$ in its *body* and all occurrences of $\mathsf{p}_1, \ldots, \mathsf{p}_k$.

For example,

$$E \equiv \mathsf{p}(\mathsf{x}, 0) \text{ where } \{\mathsf{p}(\mathsf{x}, \mathsf{y}) = \text{if } (\phi(\mathsf{x}, \mathsf{y}) = 0) \text{ then } \mathsf{y} \text{ else } \mathsf{p}(\mathsf{x}, S(\mathsf{y}))\} \tag{9}$$

is a program in the vocabulary $\{0, S, \phi, =_0\}$ of sort the sort of $\mathsf{p}$, with $\mathsf{x}$ free in its first occurrence and bound in its occurrence in the body and $\mathsf{y}$ and $\mathsf{p}$ bound in all their occurrences.

The body of a recursive program $E$ specifies a system of mutually recursive equations in the partial function variables $\mathsf{p}_1, \ldots, \mathsf{p}_k$. The denotation of a recursive program $E$ in a $\Phi$-structure $\mathcal{A}$ is obtained, intuitively, by "solving" this system and then substituting the solutions into the *head* $E_0$ of $E$.

More precisely, the parts of $E$ can be evaluated in *expansions*

$$(\mathcal{A}, p_1, \ldots, p_k) = (A, \mathbf{\Phi}, p_1, \ldots, p_k)$$

of a $\Phi$-structure $\mathcal{A}$ by arbitrary partial functions of the correct arity and sort, and they define the following *system of recursive equations* on $A$:

$$\begin{aligned} p_0(\vec{x}) &= \text{den}((\mathcal{A}, p_1, \ldots, p_k), E_0(\vec{x}, \vec{p})), \\ p_1(\vec{u}_1) &= \text{den}((\mathcal{A}, p_1, \ldots, p_k), E_1(\vec{u}_1, \vec{p})), \\ &\vdots \\ p_k(\vec{u}_k) &= \text{den}((\mathcal{A}, p_1, \ldots, p_k), E_k(\vec{u}_k, \vec{p})). \end{aligned}$$

The expressions on the right of these equations define partial functions which are *monotone* and *continuous* in their partial function arguments, and so by a standard set theoretic construction the system has a *least tuple of solutions*

$$\overline{p}_0, \overline{p}_1, \ldots, \overline{p}_k;$$

the *denotation of $E$ in $\mathcal{A}$* is then the partial function of the appropriate sort defined by the head,[11]

$$f_E^{\mathcal{A}} = \overline{p}_0 : A^n \rightharpoonup A \text{ if } \text{sort}(E_0) = \texttt{ind} \text{ and}$$

[11] In many cases, $E_0(\vec{\mathsf{x}}, \vec{\mathsf{p}}) \equiv \mathsf{p}_1(\vec{\mathsf{x}})$, so that $f_E = \overline{p}_1$, i.e., the function computed by $E$ is simply the first of the mutual fixed points of the system determined by the body of $E$.

$$f_E^{\mathcal{A}} = \overline{p}_0 : A^n \rightharpoonup \{\mathrm{T}, \mathrm{F}\} \text{ if } \mathrm{sort}(E_0) = \texttt{boole}.$$

Finally, a partial function or relation is $\mathcal{A}$-*recursive* or *recursive from* the primitives $\mathbf{\Phi}$ of $\mathcal{A}$ if it is computed in $\mathcal{A}$ by some deterministic recursive program, and we set

$$\begin{aligned}\mathbf{rec}(\mathcal{A}) =& \mathbf{rec}(A, \mathbf{\Phi})\\ =& \text{the set of all partial functions and relations which are}\\ & \text{recursive in } \mathcal{A}.\end{aligned}$$

If $S$ and Pd are the *successor* and *predecessor* functions on $\mathbb{N}$, then

$$(10)\quad f \in \mathbf{rec}(\mathbb{N}, 0, S, =) \iff f \in \mathbf{rec}(\mathbb{N}, 0, S, \mathrm{Pd}, =_0) \iff f \text{ is Turing computable.}$$

This was one of the first results about Turing computability, albeit somewhat differently formulated, and it is often used to infer that a certain $f$ is Turing computabile by giving a recursive equation or system which computes it.[12]

## 6 The Relative Recursion Thesis

It is basically trivial that *all algorithms in Section 3 compute (partial) functions or relations which are recursive from the relevant primitives*: just turn the given recursive equation into a recursive program by "formalizing" it and adding a trivial head, cf. Footnote 11.[13] There is also a rich theory of *recursion on arbitrary structures* which covers most "intuitive" definitions of algorithms and claims of computability on abstract sets from specified primitives. The examples in Sections 4 and 5 do not provide sufficient evidence that all "algorithms" from specified primitives can be expressed by

[12]For example, if $S$ is the successor on $\mathbb{N}$ and $\phi$ is total, then the program in (9) computes in $(\mathbb{N}, 0, S, \phi, =_0)$ the *minimalization* of $\phi$,

$$f_E(x) = \mu y[\phi(x, y) = 0] = \text{ the least } y \text{ such that } \phi(x, y) = 0.$$

This is the key idea in Kleene's proof that the class of Turing computable functions is closed under the minimalization operator, perhaps the earliest important connection of fixed point recursion with Turing computability.

[13]The continuous fraction algorithm operates on both reals and natural numbers and it outputs finite or infinite sequences of numbers. In the approach we are taking here, it is best viewed as an algorithm of the structure $(\mathbb{R}^+, \mathbb{N}, \mathrm{iq}, \mathrm{rem}, =_0^{\mathbb{R}}, 0^{\mathbb{N}}, S, \mathrm{Pd}, =_0^{\mathbb{N}})$ with two universes, which computes the partial function $q : \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{N} \rightharpoonup \mathbb{N}$, where $q_n = q(x, y, n)$ is the $n$'th term in the continuous fraction expansion of $\dfrac{x}{y}$, defined for all $n$ when $x$ and $y$ are not commensurable. The reduction of many-sorted recursion to recursion on one sort (other than `boole`) is routine, and so is putting down a recursive program which expresses the continuous fraction algorithm.

recursive programs, of course, but this is not the issue here; so I will leap immediately, Church-style,[14] to the strongest claim:

**The Relative Recursion Thesis**

*For every function $f : A^n \to A$ or relation $f : A^n \to \{\mathrm{T}, \mathrm{F}\}$ on a set $A$ and any set of primitives $\mathbf{\Phi}$ on $A$,*

$\mathsf{RRT}$: $f$ is computable from $\mathbf{\Phi}$

$$\iff f \text{ is recursive in the structure } \mathcal{A} = (A, \mathbf{\Phi}).$$

As with $\mathsf{CT}$, the "easy" direction ($\Leftarrow$) of $\mathsf{RRT}$ can be proved by *implementing* recursive programs using *oracles* to represent the primitives; one needs to appeal only to simple and non-controversial properties of algorithms from primitives on an arbitrary set and to assume that some basic operations on finite sequences are intuitively effective, much as Turing does for the easy direction of $\mathsf{CT}$. A proof of the non-trivial direction ($\Rightarrow$) would require showing that all computation from primitives can be reduced to *calling* (composition), *branching* and *mutual recursion*. This is possibly a simpler task than what is needed to prove $\mathsf{CT}$, but I do not see now how to go about it.[15]

## 7 Logical notions and propositions

(Tarski 1986) gave a famous explication of *logical notions*, by "applying" to logic Felix Klein's classical *Erlangen Program* for classifying geometries. We give here a (very) abbreviated and somewhat simplified version of Tarski's definitions with the aim to show that the Relative Recursion Thesis $\mathsf{RRT}$ is a logical proposition, of a very different *kind* than $\mathsf{CT}$.

**The simple type structure over a set $A$**

For every non-empty set $A$, set

$$T_0(A) = A, \qquad T_{n+1}(A) = \mathcal{P}(T_n(A)) = \text{ the set of all subsets of } T_n(A),$$

where $A$ is viewed as a set of *individuals* (atoms) with no internal set structure and every member of *the type* $T_n(A)$ is "tagged" with $n$, so that every object in these sets belongs to exactly one $T_n(A)$, $n$ being its *type*. We set

$$x \in_n y \iff x \in y \in T_{n+1},\ T^*(A) = \bigcup_n T_n(A),\ \mathbf{T}^*(A) = (T^*(A), \{\in_n\}_n).$$

[14] I have heard it said that Church claimed his version of $\mathsf{CT}$ as soon as Kleene proved that the *predecessor function* on $\mathbb{N}$ is $\lambda$-definable. Kleene took a little longer to believe it.

[15] The best arguments I know which support $\mathsf{RRT}$ come from the analysis of the notion of algorithm in (Moschovakis 1998).

This is the *simple type structure above A*.

We can use standard, set-theoretic constructions to identify in $T^*(A)$ much more complex objects than typed pure sets (of sets of sets ... of members of $A$). For example, using the Kuratowski pair,

$$x, y \in T_n(A) \implies (x, y) = \{\{x\}, \{x, y\}\} \in T_{n+2}(A),$$

which gives us the Cartesian product

$$x, y \in T_{n+1}(A) \implies x \times y = \{(u, v) \mid u \in x \ \& \ v \in t\} \in T_{n+3}(A)$$

of two sets in the same type; and iterating the process as usual, we get $k$-fold products, relations and (partial and total) functions of any arity, etc. Moreover, the embedding $x \mapsto \{x\}$ injects each $T_n(A)$ into $T_{n+1}(A)$, and its iterates give us simple embeddings

$$\jmath_n^{n+k} : T_n(A) \rightarrowtail T_{n+k}(A)$$

which "code" each $T_n(A)$ into every larger type. The upshot is that we can think of any set

$$X \subseteq T_{k_1}(A) \times \cdots \times T_{k_n}(A)$$

as a member of $T_l(A)$ for any sufficiently large $l$ and operate on these sets by the standard set operations, union, intersection, etc.[16] For example, we can code truth and falsity in $T_1(A)$ by setting

$$\mathrm{T} = A, \ \mathrm{F} = \emptyset,$$

and for any $n$, we can think of the relations

$$\in_n = \{(x, y) \mid x \in_n y\}, \quad =_n \ = \ \{(x, y) \mid x = y \in T_n(A)\}$$

as members of $T_l(A)$ for some $l$, which is not very hard to compute in this case. More significantly, for what we aim to do, fix a number $n$ and a vocabulary $\Phi = (\phi_0, \ldots, \phi_k)$ and set

$$\begin{aligned}(11)\quad \mathrm{Rec}_n(A, \Phi) = &\Big\{(f, \boldsymbol{\Phi}) \mid f : A^n \rightharpoonup A \ \& \ f \in \mathbf{rec}(A, \boldsymbol{\Phi})\Big\} \\ &\cup \Big\{(f, \boldsymbol{\Phi}) \mid f : A^n \rightharpoonup \{\mathrm{T}, \mathrm{F}\} \ \& \ f \in \mathbf{rec}(A, \boldsymbol{\Phi})\Big\},\end{aligned}$$

where $\boldsymbol{\Phi} = (\varphi_0, \ldots, \varphi_k)$ stands for any tuple of partial functions on $A$ which have the arities and sorts specified by $\Phi$ so that $(A, \varphi_0, \ldots, \varphi_k)$ is a $\Phi$-structure; we can locate (a code of) this set in $T_l(A)$, for every sufficiently large $l$ (as determined by $\Phi$ and $n$).

[16] This appeal to codings can be avoided, of course, by adopting a modern, richer definition of the type structure, with product and function types. We will not do enough in this brief note to justify the additional machinery, however, and I thought it best to stick with the simpler, classical definition.

**Logical notions**

Every permutation $\pi : A \rightarrowtail\!\!\!\!\twoheadrightarrow A$ extends naturally to a permutation $\pi^* : T_n(A) \rightarrowtail\!\!\!\!\twoheadrightarrow T_n(A)$ by the recursion

$$\pi^*(x) = \pi^*[x] = \{\pi^*(y) \mid y \in x\} \quad (x \in T_{n+1}(A)),$$

and so to $T^*(A)$; and then, easily, $\pi^*$ is an *automorphism* of the type structure $\mathbf{T}^*(A)$, i.e., it is a bijection of $T^*(A)$ with itself such that

$$x \in_n y \iff \pi^*(x) \in_n \pi^*(y) \quad (x, y \in T^*(A)).$$

Following (Tarski 1986), a set $X \in T^*(A)$ is *logical above* $A$ if it is fixed by every such automorphism, i.e.,

$$\text{for every permutation } \pi : A \rightarrowtail\!\!\!\!\twoheadrightarrow A,\ \pi^*(X) = X.$$

The motivation comes from a basic feature of definability: *if $X \in T^*(A)$ is definable* (without parameters) *in a reasonable language $\mathcal{L}$, then $X$ is fixed by every automorphism of $\boldsymbol{T}^*(A)$*—and this applies not only to the natural formal language of type theory, but to every reasonable, precisely formulated language which is naturally interpreted in $\mathbf{T}^*(A)$, including languages with second and higher order quantifiers, infinitary connectives, etc. Tarski replaces the elusive search for a characterization of "definability in some reasonable language" by a rigorous, semantic criterion which should be satisfied by all definable objects. We can then give rigorous proofs of *logicality* and *non-logicality*:

THEOREM 1 *For each $n$ and every vocabulary $\Phi$, the set* $\mathrm{Rec}_n(A, \Phi)$ *in* (11) *is logical above $A$.*

**Outline of proof.**

For any $g : A^m \rightharpoonup A$ and any permutation $\pi : A \rightarrowtail\!\!\!\!\twoheadrightarrow A$, let $g^\pi = \pi^*(g) : A^m \rightharpoonup A$ and check that

$$(12)\quad g^\pi(x_1, \ldots, x_m) = \pi g(\pi^{-1}x_1, \ldots, \pi^{-1}x_m) \quad (x_1, \ldots, x_m \in A).$$

By a simple exercise in *fixed point recursion*, for any $f, \varphi_0, \ldots, \varphi_m$,

$$(13)\quad f \text{ is recursive in } (A, \varphi_0, \ldots, \varphi_k) \iff f^\pi \text{ is recursive in } (A, \varphi_0^\pi, \ldots, \varphi_k^\pi);$$

this implies that the function part of $\mathrm{Rec}_n(A)$ is fixed by $\pi^*$, and the corresponding argument about relations finishes the proof. ■

More interesting is the next result about (intuitively understood) computability from arbitrary primitives. We label it a "Claim" rather than a theorem, because we will appeal in its proof to some assumptions about "computability from primitives", which we will not (and cannot) prove without a precise definition.

For each $n$ and each vocabulary $\Phi$, let

$$\text{(14)}\quad \mathrm{Comp}_n(A,\Phi) = \Big\{(f,\mathbf{\Phi}) \mid f : A^n \rightharpoonup A \;\&\; f \text{ is computable from } \mathbf{\Phi}\Big\} \cup \Big\{(f,\mathbf{\Phi}) \mid f : A^n \rightharpoonup \{\mathrm{T},\mathrm{F}\} \;\&\; f \text{ is computable from } \mathbf{\Phi}\Big\},$$

where $\mathbf{\Phi}$ is related to $\Phi$ as in the formulation of (11) above.

CLAIM 2 *For each $n$ and every vocabulary $\Phi$, the set* $\mathrm{Comp}_n(A,\Phi)$ *in* (14) *is logical above $A$.*

**Outline of proof**.

The key step—and where the intuitions about computability from primitives come in—is the following

*Lemma. If some process computes $f : A^n \rightharpoonup A$ from the primitives $\varphi_0,\ldots,\varphi_k$ on $A$, then for every permutation $\pi : A \rightarrowtail\!\!\!\!\twoheadrightarrow A$, the same process computes $f^\pi$ from $\varphi_0^\pi,\ldots,\varphi_k^\pi$.*

This yields

$$\text{(15)}\quad f \text{ is computable from } \varphi_0,\ldots,\varphi_k \iff f^\pi \text{ is computable from } \varphi_0^\pi,\ldots,\varphi_k^\pi,$$

from which the proof of the Claim can be completed as in Theorem 1.

*Proof of the Lemma.* Suppose $\alpha$ is some kind of process which computes a partial function $f : A^n \rightharpoonup A$ from $\varphi_0,\ldots,\varphi_k$. Our basic intuition is that for any $\vec{y} = (y_1,\ldots,y_n) \in A^n$, there is a "computation" of $\alpha$ which derives the value $f(\vec{y})$; that in the course of this computation, $\alpha$ may request from "the oracle" representing any $\varphi_i$ any particular value $\varphi_i(u_1,\ldots,u_m)$ for $u_1,\ldots,u_m$ which it has already computed from $\vec{y}$; and that if the oracles cooperate and respond to all requests, then the computation of $f(\vec{y})$ is completed in a finite number of steps. This much is probably non-controversial, and certainly true of all precisely defined "processes" (i.e., algorithms) from primitives like those in Section 3, with reasonable, precise notions of "computation". We also assume that

*the primitives $\varphi_0,\ldots,\varphi_k$ are the only non-logical operations used by $\alpha$,*

which is the most important part of our understanding of "computation from $\varphi_0, \ldots, \varphi_k$". It insures that $\alpha$ does not have access to any "hidden primitives" other than $\varphi_0, \ldots, \varphi_k$ and is again true of all standard algorithms. This supports the claim that

> if we replace the input $\vec{y}$ by $\pi(\vec{y}) = (\pi y_1, \ldots, \pi y_n)$ and also replace every request in the computation for $\varphi_i(u_1, \ldots, u_m)$ by a request for $\varphi_i^\pi(\pi u_1, \ldots, \pi u_m)$, we get a computation of $\pi f(\vec{y})$ from $\varphi_0^\pi, \ldots, \varphi_k^\pi$; and if we apply this construction to $\vec{y} = \pi^{-1}(\vec{x})$, then the output is $\pi f(\pi^{-1}(\vec{x})) = f^\pi(\vec{x})$,

which then implies the Lemma. ■

More—or less—could be put into this "proof" of Claim 2, which depends fundamentally on "appeals to intuition" and "for this reason [is] rather unsatisfactory mathematically", to use Turing's words. Or we might just *assume* Claim 2 as flowing naturally from the

> *Basic intuition*: Each value $f(\vec{x})$ of a partial function $f : A^n \rightharpoonup A$ computable from specified primitives, depends in some uniform way only on finitely many values of those primitives—and on nothing else.

We can now claim the basic result of this section:

CLAIM 3 *The Relative Recursion Thesis* RRT *is a logical proposition.*

**Proof.** Without defining *logical propositions* in general, we just assume the following which is, I think, quite plausible: if for every $m$, $X_m(A)$ and $Y_m(A)$ are logical objects over $A$, then the identity $X_m(A) = Y_m(A)$ is logical over $A$ and the universal closure

$$(\forall m)(\forall A \neq \emptyset)[X_m(A) = Y_m(A)]$$

is a logical proposition. By Theorem 1 and Claim 2, this is exactly the form of

$$\mathsf{RRT} \iff (\forall \Phi)(\forall n)(\forall A \neq \emptyset)[\mathrm{Comp}_n(A, \Phi) = \mathrm{Rec}_n(A, \Phi)]$$

once we enumerate all pairs $(n, \Phi)$ of numbers and vocabularies. ■

## 8 The punchline

The Relative Recursion Thesis restricts computability on arbitrary sets from arbitrary, specified primitives and does not say anything directly about (absolute) computability or recursion on the natural numbers. However:

CLAIM 4 *A function* $f : \mathbb{N}^n \to \mathbb{N}$ *or relation* $f : \mathbb{N}^n \to \{\mathrm{T}, \mathrm{F}\}$ *is computable if and only if* $f$ *is computable on* $\mathbb{N}$ *from* $0, S, =$

— because $(\mathbb{N}, 0, S, =)$ *is just what the natural numbers are.*

This is also a Claim rather than a Theorem, because it is grounded on an assumption about the nature of natural numbers—*what they are*—which cannot be proved any more than CT or RRT can be proved.

Very briefly, about a problem which has been discussed as extensively as any other in the Philosophy of Mathematics since the 1870s, there are two basic facts about the natural numbers, both due to Frege and (mostly) Dedekind:

(i) $(\mathbb{N}, 0, S, =)$ is a *Peano system*, i.e., $0 \in \mathbb{N}$, the successor function is a bijection of $\mathbb{N}$ with its non-0 elements, and the *Induction Axiom* holds, i.e., for every $X \subseteq \mathbb{N}$,[17]

$$\Big(0 \in X \ \&\ (\forall x \in X)[S(x) \in X]\Big) \implies X = \mathbb{N}.$$

(ii) *Dedekind's Theorem*: Any two Peano systems are (uniquely) isomorphic.

These lead to what I will call

**The Standard View**. The natural numbers are a Peano system—*and that is all they are.*

There are many well-known and much discussed problems with the claim that the numbers are *just* a Peano system, some of them stemming from the fact that there is no natural way to "select" a particular (privileged) one, cf. (Benacerraf 1965). At the same time, there are also many responses to this problem, e.g., structuralist or modal approaches, which get around the problem in various (sometimes very sophisticated) ways. I do think, however, that the common, starting point for all philosophical views about the natural numbers are (i) and (ii). This is what I am trying to convey by saying "*and that is all the numbers are*": the idea is that if we derive some results about the numbers using only the fact that they are a Peano system, then these results will find a natural expression in any coherent approach to the foundations of number theory. This should apply to Claim 4, which can then be used in the proof of the following

THEOREM 5 *The Relative Recursion Thesis and the Standard View about numbers imply the Church-Turing Thesis, i.e.,*

$$\mathsf{RRT} + \text{the Standard View} \implies \mathsf{CT}.$$

[17] The equality relation $=$ is not usually included in the definition of a Peano system, but it is implicit in the definition of *isomorphism*.

**Proof.** For the non-trivial direction of CT, suppose $f : \mathbb{N}^n \to \mathbb{N}$ is computable; $f$ is then computable from $0, S, =$ by the Standard View; and so it is recursive in $(\mathbb{N}, 0, S, =)$ by RRT; and so it is Turing computable by (10). ■

### 8.1 Concluding remarks

Theorem 5 suggests that the meaning and truth value of the Church-Turing Thesis do not depend on any deep properties of numbers or any assumptions about "the primitives of computation" or whether "all computation is symbolic"; these are now replaced by a well understood view of "what the numbers are". It does not prove CT, because RRT is not immediate: its meaning and justification require identifying a basis for the *logical primitives of computation on an arbitrary set from arbitrary functions and relations*. They should be composition, branching and recursion, of course, but it is not obvious to me how to prove this beyond a reasonable doubt.

## Bibliography

Benacerraf, P. (1965). What numbers could not be. *Philosophical Review*, *74*, 47–73, reprinted in Philosophy of Mathematics, Selected readings, eds. Benacerraf, P. and Putnam, H., Cambridge University Press, 1983.

Church, A. (1935). An unsolvable problem in elementary number theory. *Bulletin of the American Mathematical Society*, *41*, 332–333, this is an abstract of Church (1936*b*).

Church, A. (1936*a*). A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, *1*(1), 40–41, doi:10.2307/2269326.

Church, A. (1936*b*). An unsolvable problem in elementary number theory. *American Journal of Mathematics*, *58*(2), 345–363, an abstract of this paper was published in Church (1935).

Dershowitz, N. & Gurevich, Y. (2008). A natural axiomatization of computability and proof of Church's Thesis. *The Bulletin of Symbolic Logic*, *14*, 299–350, doi:10.2178/bsl/1231081370.

Gandy, R. (1980). Church's Thesis and principles for mechanisms. In *The Kleene Symposium*, Barwise, J., Keisler, H. J., & Kunen, K., eds., Amsterdam; New York: North Holland Publishing, 123–148.

Gandy, R. (1995). Church's Thesis and principles for mechanisms. In *The Kleene Symposium*, Barwise, J., Keisler, H. J., & Kunen, K., eds., North Holland Publishing Co., 123–148.

Gurevich, Y. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Transactions on computational logic*, *1*, 77–111.

Kleene, S. C. (1952). *Introduction to Metamathematics*. New York: Van Nostrand; North Holland.

Kripke, S. A. (2000). From the Church-Turing Thesis to the First-Order Algorithm Theorem. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, LICS '00, Washington, DC, USA: IEEE Computer Society, URL `http://dl.acm.org/citation.cfm?id=788022.789011`, the reference is to an abstract. A video of a talk by Saul Kripke at *The 21st International Workshop on the History and Philosophy of Science* with the same title is posted at `www.youtube.com/watch?v=D9SP5wj882w`, and this is my only knowledge of this article.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, Braffort, P. & Herschberg, D., eds., Amsterdam: North-Holland, 33–70.

Moschovakis, Y. N. (1968). Review of four papers on Church's Thesis. *Journal of Symbolic Logic*, *33*, 471–472.

Moschovakis, Y. N. (1989). The formal language of recursion. *Journal of Symbolic Logic*, *54*(4), 1216–1252, doi:10.1017/S0022481200041086.

Moschovakis, Y. N. (1998). On founding the theory of algorithms. In *Truth in Mathematics*, Dales, H. G. & Oliveri, G., eds., Oxford: Clarendon Press, 71–104, posted on `www.math.ucla.edu/~ynm/ papers/foundalg.pdf`.

Sieg, W. (2002). Calculations by man and machine: mathematical presentation. In *In the Scope of Logic, Methodology and Philosophy of Science*, Gärdenfors, P., Woleński, J., & Kijania-Placek, K., eds., Dordrecht; Boston: Kluwer Academic Publishers, 247–262.

Tarski, A. (1951). A decision method for elementary algebra and geometry. RAND Corporation report. Prepared for publication with the assistance of J.C.C. McKinsey.

Tarski, A. (1986). What are logical notions? *History and Philosophy of Logic*, *7*, 143–154, doi:10.1080/01445348608837096, edited by John Corcoran.

Tiuryn, J. (1989). A simplified proof of DDL $<$ DL. *Information and Computation*, *82*, 1–12.

Tucker, J. & Zucker, J. (2000). Computable functions and semicomputable sets on many-sorted algebras. In *Handbook of Logic in Computer Science*, vol. 5, Abramsky, S., Gabbay, D., & Maibaum, T., eds., New York: Oxford University Press, 317–523.

Turing, A. M. (1936). On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *42*, 230–265, *A correction*, ibid. volume 43 (1937), pp. 544–546.

van den Dries, L. & Moschovakis, Y. N. (2004). Is the Euclidean algorithm optimal among its peers? *Bulletin of Symbolic Logic*, *10*(3), 390–418.

Yiannis N. Moschovakis
Department of Mathematics, University of California, Los Angeles
USA
ynm@math.ucla.edu
Department of Mathematics, University of Athens
Greece