The Polynomial-Time Hierarchy

by Edward Nelson
Department of Mathematics
Princeton University
http://www.math.princeton.edu/~nelson/pt.html

These are notes for a course (MAT 504, Spring 1998) on the polynomial-time hierarchy, using mathematical logic and nonstandard analysis.

Information of all kinds is stored in a computer by strings of bits. A set p of strings presents a computational problem: given a string x, to determine whether it is in p. It is a polynomial-time problem in case there exist a computer program and a polynomial π such that for any string x the program prints out "yes" in a number of steps less than $\pi(|x|)$ if and only if x is in p. (Here $|x|$ denotes the length of x.) With some caveats, these are the computational problems whose solution is feasible. A similar definition is made for problems involving pairs, triples, etc., of strings, and P is the set of all polynomial-time problems.

Let q be a polynomial-time problem in a pair (x, y) of strings and let π be a polynomial. Consider the following problem p: given a string x, to determine whether there exists a string y with $|y| \le \pi(|x|)$ such that (x, y) is in q. These are the NP problems. The dual problem, to determine whether for all such y the pair (x, y) is not in q, is a $\mathsf{co\text{-}NP}$ problem. A brute force approach to an NP problem involves a search through all strings y with $|y| \le \pi(|x|)$. The number of such strings grows exponentially, so such a method is infeasible. But there may be better algorithms than an exhaustive search and the question arises as to whether indeed $\mathsf{P} \ne \mathsf{NP}$.

The sets P, NP, and $\mathsf{co\text{-}NP}$ are the lowest levels of the polynomial-time hierarchy. Let $\Sigma_0^{\mathrm{p}} = \Pi_0^{\mathrm{p}} = \mathsf{P}$, let $\Sigma_1^{\mathrm{p}} = \mathsf{NP}$, and let $\Pi_1^{\mathrm{p}} = \mathsf{co\text{-}NP}$. The higher levels are defined by recursion. Suppose that Σ_ν^{p} and Π_ν^{p} have been defined. Let q be in Π_ν^{p}, a problem in a pair (x, y) of strings, and let π be a polynomial. Consider the following problem p: given a string x, to determine there exists a string y with $|y| \le \pi(|x|)$ such that (x, y) is in q. These are the $\Sigma_{\nu+1}^{\mathrm{p}}$ problems, and the $\Pi_{\nu+1}^{\mathrm{p}}$ problems are their duals. One says that the polynomial-time hierarchy does not collapse in case all of these sets are distinct, so that in particular $\mathsf{P} \ne \mathsf{NP}$.

The polynomial-time hierarchy is complicated because its base P is complicated. Our first goal is to construct a simpler hierarchy, called the basic hierarchy, and show that the polynomial-time hierarchy collapses if and only if the basic hierarchy collapses.

1. Strings

Let W be a set and let $\lambda \in \mathbb{N}$. A *λ-ary function on W* is a function

$$f\colon W^\lambda \to W$$

and a *λ-ary predicate on W* is a subset

$$p \subseteq W^\lambda.$$

We call λ the *index*.

An element of W^λ is a W-valued function $\mu \mapsto \xi_\mu$ defined for $1 \le \mu \le \lambda$. It is often convenient to index by variables rather than by the numbers from 1 to λ. Let $\mathcal{V}$ be a countably infinite set whose elements are called *variables*. Let Φ be a finite set of variables; then an element ξ of W^Φ is a W-valued function $x \mapsto \xi_x$ defined for x in Φ. A *Φ-function on W* is a function

$$f\colon W^\Phi \to W$$

and a *Φ-predicate on W* is a subset

$$p \subseteq W^\Phi.$$

A *W-function* is a Φ-function for some finite set Φ of variables, and similarly for a *W-predicate*. The distinction between a λ-ary function and a Φ-function, where Φ is a set of λ variables, is one of notational convenience; one can pass back and forth by ordering the variables. Sometimes by abuse of notation we write $f(x_1, \ldots, x_\lambda)$, where $x_1, \ldots, x_\lambda$ are in W, for the value of a W-function; then it is understood that there are variables $\tilde{x}_1, \ldots, \tilde{x}_\lambda$ with $\xi_{\tilde{x}_1} = x_1, \ldots, \xi_{\tilde{x}_\lambda} = x_\lambda$.

For p a Φ-predicate, define its *complement* $\complement p$ by

$$\complement p = W^\Phi \smallsetminus p.$$

Then $\complement p$ is also a Φ-predicate.

For p_1 a Φ_1-predicate and p_2 a Φ_2-predicate, define their *union* $p_1 \cup p_2$ by

$$p_1 \cup p_2 = \{\, \xi \in W^{\Phi_1 \cup \Phi_2} : \xi_1 \in p_1 \text{ or } \xi_2 \in p_2 \,\}$$

where ξ_1 and ξ_2 are the restrictions of ξ to Φ_1 and Φ_2. Then $p_1 \cup p_2$ is a $(\Phi_1 \cup \Phi_2)$-predicate. We call $\complement$ and $\cup$ *semantic connectives*. Other semantic connectives can be defined in terms of them, including the *intersection* $p_1 \cap p_2 = \complement(\complement p_1 \cup \complement p_2)$.

For p a Φ-predicate and x a variable, define the *projection* $\Sigma x\, p$ by

$$\Sigma x\, p = \{\, \xi \in W^{\Phi \setminus \{x\}} : \ \xi \text{ has an extension in } p \,\}.$$

Then $\Sigma x\, p$ is a $(\Phi \setminus \{x\})$-predicate. The *co-projection* is defined by $\Pi x\, p = \complement \Sigma x \complement p$. We call Σx and Πx, which are mappings of predicates to predicates, *semantic quantifiers* (*existential* and *universal* respectively).

Let $\phi\colon \Phi_1 \to \Phi_2$. If p is a Φ_1 predicate, then $\{\, \xi \in W^{\Phi_2} : \phi \circ \xi \in p \,\}$ is a Φ_2 predicate obtained from p by *replacement of variables.*

A *base* is a set of W-predicates closed under the semantic connectives and replacement of variables. Let B be a base. Then the *unbounded hierarchy with base B* is defined by recursion as follows. Let

$$\Sigma_0(B) = \Pi_0(B) = B.$$

If $\Sigma_\nu(B)$ and $\Pi_\nu(B)$ have been defined, let $\Sigma_{\nu+1}(B)$ be the set of all predicates of the form

$$\Sigma y_1 \ldots \Sigma y_\lambda\, q$$

where $q \in \Pi_\nu(B)$, and let $\Pi_{\nu+1}(B)$ be the set of all $\complement p$ such that $p \in \Sigma_{\nu+1}(B)$.

Let $\xi \mapsto |\xi|$ be a function from W to $\mathbb{N}$. Let p be a Φ-predicate, x a variable, and π a polynomial, with coefficients in $\mathbb{N}$, in the $|\xi_y|$ for y in $\Phi \setminus \{x\}$. Then we define

$$\Sigma|x|{\le}\pi\, p = \{\, \xi \in W^{\Phi \setminus \{x\}} : \xi \text{ has an extension in } p \text{ with } |\xi_x| \le \pi \,\}.$$

We define $\Pi|x|{\le}\pi\, p = \complement\Sigma|x|{\le}\pi\, \complement p$.

Let B be a base. Then the *polynomially bounded hierarchy with base B* is defined by recursion as follows. Let

$$\Sigma^{\mathrm{p}}_0(B) = \Pi^{\mathrm{p}}_0(B) = B.$$

If $\Sigma^{\mathrm{p}}_\nu(B)$ and $\Pi^{\mathrm{p}}_\nu(B)$ have been defined, let $\Sigma^{\mathrm{p}}_{\nu+1}(B)$ be the set of all predicates of the form

$$\Sigma|y_1|{\le}\pi \ldots \Sigma|y_\lambda|{\le}\pi\, q$$

where $q \in \Pi^{\mathrm{p}}_\nu(B)$, and let $\Pi^{\mathrm{p}}_{\nu+1}(B)$ be the set of all $\complement p$ such that $p \in \Sigma^{\mathrm{p}}_{\nu+1}(B)$.

A *bit* is 0 or 1. Let W be the set of all finite sequences of bits. An element x of W is called a *string* and its length is denoted by $|x|$. We call W-functions and W-predicates *string functions* and *string predicates*.

The set W^0 has one element, $\emptyset$, so there are two string predicates of index 0, $\mathsf{T} = \mathsf{W}^0$ (*true*) and $\mathsf{F} = \emptyset$ (*false*). We identify a 0-ary string function with its value, a string. Let ϵ be the empty string. We identify sequences of length one with their value, so 0 and 1 are strings.

Some authors use $\mathbb{N}$ rather than W to discuss hierarchies. The difference is a minor one of notation since the set of binary representations of natural numbers differs inessentially from W.

The definition of string predicate is a formalization of the intuitive notion of an objective property of strings, one that is not a matter of opinion. Certain strings must be avoided because they disparage beef and if you utter such a string you may be sued in the state of Texas. But it seems implausible that "x disparages beef" can be formalized as a string predicate.

The first question about a string function or predicate is whether it can be calculated by some algorithm. This is formalized by requiring that it be calculable by a Turing machine, in which case it is *recursive*. Let R be the set of recursive predicates on W. The unbounded hierarchy with base R is the *arithmetical hierarchy*, and $\Sigma_1(\mathsf{R})$ is the set of recursively enumerable predicates; this is known not to be equal to R. In fact, it is known that the arithmetical hierarchy does not collapse: all the sets in the hierarchy are distinct.

The next question about a string function or predicate is whether it can be calculated by a feasible algorithm. This is formalized by requiring that it be calculable by a Turing machine in a number of steps bounded by a polynomial in the lengths of the arguments, in which case it is *polynomial-time*. Let P be the set of polynomial-time predicates on W. The polynomially bounded hierarchy with base P is the *polynomial-time hierarchy*, and NP is another name for $\Sigma_1^{\mathrm{p}}(\mathsf{P})$.

Let $x \oplus y$, also denoted by xy and called the *concatenation* of x and y, be the string of length $|x| + |y|$ consisting of the bits of x followed by the bits of y. Let $x \odot y$, the *zero product* of x and y, be the string of length $|x| \cdot |y|$ all of whose bits are 0. Let $x \preceq y$ (that is, $(x, y) \in \preceq$), read as *x is shorter than y*, if and only if $|x| \leq |y|$. Also, *equality* $x = y$ is string predicate. We call ϵ, 0, 1, $\oplus$, and $\odot$ the *basic string functions*, and $\preceq$ and $=$ are the *basic string predicates*.

The next three sections present important results on P *and* NP*, but they will not be used in the sequel.*

2. Register machines

Turing machines were introduced in 1936 [Tu]. In reading this paper one must remember that a "computer" is one who computes. The discussion here is based on Leivant's paper [Le] and we construct a special case of his register machines. These are a convenient variant of Turing machines.

The natural numbers $\mathbb{N}$ are generated from the constant 0 by repeatedly applying the successor function S. On W we have two *successor functions*, the unary string functions

$$\begin{aligned} \mathrm{S}_0 &: x \mapsto 0x \\ \mathrm{S}_1 &: x \mapsto 1x, \end{aligned}$$

and W is generated by repeatedly applying them to ϵ. The *constructors* are the 0-ary string function ϵ and the successor functions. The *predecessor* (or destructor) P is the unary string function satisfying $\mathrm{P}\epsilon = \epsilon$, $\mathrm{PS}_0 x = x$, and $\mathrm{PS}_1 x = x$. The *left bit* L is the unary string function satisfying $\mathrm{L}\epsilon = \epsilon$, $\mathrm{LS}_0 x = 0$, and $\mathrm{LS}_1 x = 1$.

Let the *states* be a countably infinite set containing the *initial state* s_1 and the *terminal state* s_0, where $s_1 \neq s_0$. Let $x \mapsto \rho(x)$ be a bijection from the variables to a set of *registers*. A *command* is one of the following:

constructor command	(s, r, c, s')
destructor command	(s, r, s')
branch command	(s, r, s', s'', s''')

where the s's are states, r is a register, and c is 0 or 1. Notice that each command has a unique register associated with it.

A *register machine* M is a triple $M = (S, R, C)$ where S is a finite set of states containing s_1 and s_0, R is a finite set of registers, and C is a finite set of commands, with the states and registers in the commands of C in S and R, and such that for each s in $S \setminus \{s_0\}$ there is a unique command in C whose first element is s.

Let M be a register machine over W. A *configuration* of M is a pair $\kappa = (s, F)$ where $s \in S$ and $F: R \to \mathsf{W}$. Thus a configuration specifies the state and the contents of the registers.

We define the *step function*

$$\sigma: (s, F) \mapsto (s^*, F^*)$$

mapping configurations to configurations as follows. For the terminal state we let $\sigma(s_0, F) = (s_0, F)$. Consider a non-terminal state s. There is a unique command in C whose first element is s and a unique register r in that command. We set $F^*(r') = F(r')$ for $r' \neq r$, so we need only specify $F^*(r)$ and the state s^*. We do this as follows:

$$\begin{array}{lll}
\text{constructor } (s,t,c,s') : & F^*(r) = \mathrm{S}_c F(r), & s^* = s' \\
\text{destructor } (s,r,s') : & F^*(r) = \mathrm{P}F(r), & s^* = s' \\
\text{branch } (s,r,s',s'',s''') : & F^*(r) = F(r), & s^* = \begin{cases} s', & \mathrm{L}F(r) = \epsilon \\ s'', & \mathrm{L}F(r) = 0 \\ s''', & \mathrm{L}F(r) = 1. \end{cases}
\end{array}$$

Let M be a register machine and let Φ be such that $\rho(\Phi) \subseteq R$. Let $\xi \in \mathsf{W}^\Phi$. The *initial configuration* is (s_1, F_1) where $F_1(r) = \xi_x$ when $r = \rho(x)$ with x in Φ and otherwise is ϵ. The *computation* of M on ξ is the sequence $\nu \mapsto \sigma^\nu(s_1, F_1)$. The computation *halts* in case for some ν and F_0 we have $\sigma^\nu(s_1, F_1) = (s_0, F_0)$, in which case the last register r_0 accessed (the register associated to the command in the penultimate configuration) is called the *output register* of the computation and $F_0(r_0)$ is the *output*.

A string function is *recursive* in case there exists a register machine M such that the computation always halts with output the value of the function. It is *polynomial-time* in case the number of steps in the computation is bounded by a polynomial in the lengths of the arguments of f, and it is of *degree* δ in case the number of steps in the computation is bounded by a polynomial of degree δ in the lengths of the arguments of f. Identifying string predicates with their characteristic functions and requiring the output to be 0 (false) or 1 (true), we obtain the set R of *recursive predicates*, the set P of *polynomial-time predicates*, and the set P_δ of *degree* δ *predicates*.

We have described what is essentially a deterministic register machine over W in the sense of [Le, §3.1]. One change, following the suggestion in footnote 6, is to use the same register for input and output of the commands. This has the effect that each step of a computation changes at most one bit of at most one register.

Now the sets $\Sigma^{\mathrm{p}}_\nu(\mathsf{P})$ and $\Pi^{\mathrm{p}}_\nu(\mathsf{P})$ of the polynomial-time hierarchy are well defined. I have avoided introducing non-deterministic machines and oracles. The definition of the polynomial-time hierarchy given here is based on Wrathall's characterization [Wr] [GaJo, §7.1] [Bu, §1.4] rather than the original definition by Meyer and Stockmeyer [MeSt] [St].

The basic string functions and string predicates are polynomial-time. To prove this is a straightforward exercise in programming register machines. The constants are of degree 0, $\oplus$ is of degree 1, $\odot$ is of degree 2, and $\preceq$ and $=$ are of degree 1.

3. Ramified recurrence

We shall introduce Leivant's notion [Le] of ramified recurrence.

We begin with ordinary recurrence, also called recursion. (Since it does not suffice to give all recursive functions, Leivant's "recurrence" is preferable.) First, two preliminaries.

Let g be a Φ-function and for each x in Φ let h_x be a Φ_x-function for some Φ_x. Let $\Psi = \bigcup_{x\in\Phi} \Phi_x$ and define the *composition* f of g with the h_x to be the Ψ-function defined as follows. Let ξ be in W^Ψ. For each x in Φ, let $\xi|_{\Phi_x}$ be the restriction of ξ to Φ_x. Then $x \mapsto h_x(\xi|_{\Phi_x})$ is an element η of W^Φ; let $f(\xi) = g(\eta)$.

Next let $\Phi' \supseteq \Phi$ and let g be a Φ-function. Define the Φ'-function f, obtained from g by *expansion of variables*, by letting $f(\xi) = g(\xi|_\Phi)$.

Now let $\vec{x}$ be a finite set of variables and let y, the *recurrence variable*, and the α-tuple $\vec{z}$, the *critical variables*, be distinct variables not in $\vec{x}$. Let $\vec{g}_\epsilon(\vec{x})$, $\vec{g}_0(\vec{z}, y, \vec{x})$, and $\vec{g}_1(\vec{z}, y, \vec{x})$ be α-tuples of given functions. Then

$$\begin{cases} \vec{f}(\epsilon, \vec{x}) = \vec{g}_\epsilon(\vec{x}) \\ \vec{f}(\mathrm{S}_0 y, \vec{x}) = \vec{g}_0\big(\vec{f}(y, \vec{x}), y, \vec{x}\big) \\ \vec{f}(\mathrm{S}_1 y, \vec{x}) = \vec{g}_1\big(\vec{f}(y, \vec{x}), y, \vec{x}\big) \end{cases}$$

is a definition of an α-tuple $\vec{f}$ of functions by *recurrence*. We allow some of the variables to be missing; that is, the given functions may be defined by expansion of variables from functions depending on fewer variables.

If the critical variable is missing—that is, if $\vec{g}_0(\vec{z}, y, \vec{x}) = \vec{h}_0(y, \vec{x})$ and $\vec{g}_1(\vec{z}, y, \vec{x}) = \vec{h}_1(y, \vec{x})$ for some functions $\vec{h}_0$ and $\vec{h}_1$—the recurrence is called *flat*. The destructor can be defined by flat recurrence:

$$\begin{cases} \mathrm{P}(\epsilon) = \epsilon \\ \mathrm{P}(\mathrm{S}_* y) = y. \end{cases}$$

When we write S_* we mean that $g_0 = g_1$, so the same equation applies to both $\mathrm{S}_0 y$ and $\mathrm{S}_1 y$. Here $g_\epsilon = \epsilon$ and $g_0(z, y) = g_1(z, y) = y$.

Here is another useful function defined by flat recurrence:

$$\begin{cases} \mathrm{Case}\,(\epsilon, a, b, c) = a \\ \mathrm{Case}\,(\mathrm{S}_0 y, a, b, c) = b \\ \mathrm{Case}\,(\mathrm{S}_1 y, a, b, c) = c. \end{cases}$$

Here the recurrence variable is missing as well as the critical variable.

The set PR of *primitive recursive functions* is the smallest set of functions containing the constructors and closed under composition and recurrence. (When we say "closed under composition" we include expansion of variables.)

Every primitive recursive function is recursive. We sketch a proof to introduce some constructs that will be used in the next section.

The machine $\mathrm{Rev}(r; r', r'')$ with commands

$$\begin{array}{lll} (s_1, r, s_0, s_2, s_3) & & \\ (s_2, r, s_4) & (s_4, r', 0, s_6) & (s_6, r'', 0, s_1) \\ (s_3, r, s_5) & (s_5, r', 1, s_7) & (s_7, r'', 1, s_1) \end{array}$$

puts the contents $F(r)$ of r in reverse order to the left of $F(r')$ and $F(r'')$, leaving r empty, in $4 \cdot |F(r)| + 1$ steps. If r'' is omitted with s_6 and s_7 replaced by s_1, we obtain a machine $\mathrm{Rev}(r; r')$. Let r' and r'' be empty. Then $\mathrm{Rev}(r; r'')$ followed by $\mathrm{Rev}(r''; r, r')$ produces a copy of $F(r)$ in r' with the contents of r unchanged; we call this a *copy*.

Any machine can be altered as follows: first copy the contents of the input registers onto *incoming registers* and at the end of the computation copy the output onto an *outgoing register* fixed in advance; call this a *copy machine*.

To compute the composition $f(\vec{x}) = g\big(\vec{x}, \vec{h}(\vec{x})\big)$, let the outgoing registers of the $\vec{h}$ be identical with the appropriate incoming registers of g.

To perform a recursion with recurrence variable y and critical variables $\vec{z}$, let r be the incoming register for y, reverse $F(r)$ onto r' leaving the contents of r unchanged, and let r'' be empty. First compute $\vec{g}_\epsilon(\vec{x})$, storing the results in $\vec{\rho}(\vec{z})$. Then do the following loop. If r' is empty, the recursion is complete. If the first bit of r' is 0, delete it from r', put it to the left of $F(r'')$, and compute $\vec{g}_0$ on the contents of r'', $\vec{\rho}(\vec{z})$, and the incoming registers for $\vec{x}$, storing the result in $\vec{\rho}(\vec{z})$, and continue the loop, proceeding similarly if the first bit of r' is 1.

In this way we construct a canonical copy machine for each construction of a primitive recursive function and thereby prove, as claimed, that every primitive recursive function is recursive.

Now we construct $(y, x) \mapsto y \oplus x$ by recurrence:

$$\begin{cases} \epsilon \oplus x = x \\ \mathrm{S}_0 y \oplus x = \mathrm{S}_0(y \oplus x) \\ \mathrm{S}_1 y \oplus x = \mathrm{S}_1(y \oplus x). \end{cases}$$

Here $g_\epsilon(x) = x$, $g_0(z, y, x) = S_0 z$, and $g_1(z, y, x) = S_1 z$.

We define *exponentiation* $\uparrow$, a unary function, by recurrence from concatenation:

$$\begin{cases} \uparrow\epsilon = \mathrm{S}_0\epsilon \\ \uparrow\mathrm{S}_* y = \uparrow y \oplus \uparrow y. \end{cases}$$

Then $\uparrow y$ consists of $2^{|y|}$ bits 0.

Concatenation and exponentiation are primitive recursive, but concatenation is feasible and exponentiation is infeasible. How do we tell good recurrences from bad? To understand ramified recurrence it is necessary critically to examine our notion of what a number is, and to that end I shall tell a fable.

Once there was an encounter between a Mathematician and a Primitive Person and a conversation ensued. The Mathematician spoke first:

Greetings, friend.

Greetings.

Do you know what a number is?

No.

Do any in your tribe keep sheep?

Yes; most of us are shepherds.

Are records kept of the sheep?

I am the Record Keeper. For each shepherd I have a clay tablet with a tally mark / for each sheep. For example, I can tell you that Kochen has /// sheep and Trotter has // sheep.

The notion of number is an abstract notion derived from the example of tally marks on a clay tablet. We start with the number 0, represented by a blank tablet, and generate numbers by the successor function, represented by making a new tally mark. Each time we do this we obtain a number different from its predecessor, and there is no end to the process.

That is admirably clear.

I know many things about numbers. For example, suppose that a number can be written in a square array on a tablet, even an enormous tablet, and you make a copy of the tablet. Then the tally marks on the pair of tablets cannot be arranged in a square.

How can you possibly know this about all numbers, since as you say there is no end to the process of constructing numbers?

Let me explain. I must introduce some concepts. Here is an example of addition. Put Kochen's and Trotter's tablets side by side. See, together they have ///// sheep.

Addition is represented by putting the tablets together.

Right! Now let me explain doubling. If x is a number, then the double of x is x added to x.

I think I see. In fact the double of x is the same as x, right?

Why do you say that?

Consider the example that x is the number of Kochen's sheep. Kochen has $///$ sheep; Kochen and Kochen together has $///$ sheep.

No, you put two tablets together to add.

What two tablets? There is only one tablet for Kochen's sheep.

But you could copy it, couldn't you? Please do so.

All right. Here is a copy, with $///$ on it. I put the tablets together and find $//////$, so the double of $///$ is $//////$. I understand now. If x is a number, then the double of x is a number provided that x can be doubled.

But a number can always be doubled.

How do you know?

Well, 0 can be doubled. If x can be doubled, then the successor of x can be doubled, because this only involves doubling x and then adjoining $//$.

I agree.

Therefore all numbers can be doubled.

Why?

By induction, don't you see? *Aside*: This person is so primitive as to doubt that the doubling function is total!

The Primitive Person led the Mathematician to the entrance of a cave. Stepping inside, they saw a vast row of clay tablets stretching into the indiscernible depths of the cave. The Primitive Person said:

Here are all the records from the days of the First Record Keeper, kept for generations beyond memory. These tablets together represent a number N. You must convince me that N can in principle be doubled.

Just apply the inductive reasoning you agree on. Go to the beginning of the records and apply this argument step by step for each tally mark in the cave.

You are begging the question; I do not know that it is possible even in principle to reach the beginning of the records.

Now the Mathematician realized that the Primitive Person's intuitive concept of number included a concrete example N for which the person was not convinced that N is standard, and tried a different approach:

It seems to me that your concept of number is somewhat vague.

It is.

I have a clearer concept, which I shall call natural number. Let me explain it to you.

Please do.

Start with numbers in your sense. Consider properties of numbers. Call a property inductive in case it holds for 0 and whenever it holds for a number, it holds for the successor of that number. Consider the property of being finite, meaning that every inductive property holds. Then finite is an inductive property. A natural number is a number with the property of being finite.

Since having a double is an inductive property, every natural number has a double that is a natural number.

I am lost. We start with my vague concept of number. You say that a natural number is a vague number with a certain property (being finite) that a vague number may have. Your definition of this property depends on a certain notion (inductive) which appears to be a property of properties of vague numbers. I must confess that your concept of number seems to me vaguer than mine; in fact, it makes no sense to me at all.

At this point the Mathematician made a mistake. The wise response would have been, "Unless you understand this kind of reasoning, you will never understand what a number is." Instead, the Mathematician said:

Very well. Let's consider the property of having a double. Is that clear?

Yes.

Call a number a stage / number in case it has a double. This new notion is just as good, because 0 is a stage / number and if x is a stage / number, so is its successor.

Agreed. But the double of a stage / number might not be a stage / number.

Wait. Define a stage // number to be a stage / number whose double is a stage / number, and so on. For each stage y number, its double is a stage predecessor of y number. Now define a doublable number to be a number that is a stage y number for all y. Clearly, if x is a doublable number, then the double of x is also a doublable number.

Yes, but I don't see that 0 is a doublable number. For example, is 0 a stage N number?

Forget N! I'm talking about stage y numbers only when y is a number!

Do you know what a number is?

No.

Goodbye, friend.

Goodbye.

The Primitive Person went back to making marks on clay. The Mathematician went back to making marks on paper.

There is no problem with $\mathbb{N}$: it is a constant in an extension by definitions of the first-order theory ZFC, Zermelo-Fraenkel set theory with the axiom of choice. But the metamathematical concept of the totality of all natural numbers is indeed a vaguer notion than the original concept of counting numbers. It requires a reification of properties and properties of properties, confounding potentialities with actualities. If we think we have unambiguously specified the natural numbers, someone may come along, like Dostoyevsky's man with arms akimbo, and proclaim that our notion is too inclusive, containing nonstandard numbers as well as standard ones—and who can gainsay him?

Fix a nonstandard number N and call a string *stage* 0 in case its length is less than N plus some standard number. Then the stage 0 strings contain ϵ and are closed under the successors, but not under concatenation. Call a string Y *stage* $\oplus$ in case for all stage 0 strings x, the concatenation $Y \oplus x$ is a stage 0 string. Then the stage $\oplus$ strings also contain ϵ and are closed under the successors; they are not closed under concatenation but the concatenation of two stage $\oplus$ strings—more generally, of a stage $\oplus$ string and a stage 0 string—is a stage 0 string.

This lengthy discussion was to motivate the notion of ramified recurrence. Let W_0 and W_1 be disjoint copies of W. We introduce variables of two *sorts*, with sort 0 variables ranging over W_0 and sort 1 variables ranging over W_1. A function from $\mathsf{W}_1^{\Phi_1} \times \mathsf{W}_0^{\Phi_0}$, where Φ_1 and Φ_0 are finite sets of variables of sort 1 and sort 0 respectively, to W_0 is called a *sorted function*. Composition of sorted functions must respect sorts. A recurrence is *ramified* in case the critical variable, if present, is of sort 0 and the recurrence variable is of sort 1. Flat recurrences are ramified since the critical variable is not present.

For this discussion, but only in this and the following section, let us convene that upper case is used for variables of sort 1 and lower case for variables of sort 0. Then a non-flat ramified recurrence is of the form

$$\begin{cases} \vec{f}(\epsilon, \vec{X}, \vec{x}) = \vec{g}_\epsilon(\vec{X}, \vec{x}) \\ \vec{f}(\mathrm{S}_0 Y, \vec{X}, \vec{x}) = \vec{g}_0\big(\vec{f}(Y, \vec{X}, \vec{x}), Y, \vec{X}, \vec{x}\big) \\ \vec{f}(\mathrm{S}_1 Y, \vec{X}, \vec{x}) = \vec{g}_1\big(\vec{f}(Y, \vec{X}, \vec{x}), Y, \vec{X}, \vec{x}\big) \end{cases}$$

where the critical variables $\vec{z}$ in $\vec{g}_0(\vec{z}, Y, \vec{X}, \vec{x})$ and $\vec{g}_1(\vec{z}, Y, \vec{X}, \vec{x})$ are sort 0.

As an example, we construct concatenation as a sorted function

$$\oplus \colon \mathsf{W}_1^{\{Y\}} \times \mathsf{W}_0^{\{x\}} \to \mathsf{W}_0$$

by a ramified recurrence as follows:

$$\begin{cases} \epsilon \oplus x = x \\ \mathrm{S}_0 Y \oplus x = \mathrm{S}_0(Y \oplus x) \\ \mathrm{S}_1 Y \oplus x = \mathrm{S}_1(Y \oplus x). \end{cases}$$

Here $g_\epsilon(x) = x$, $g_0(z, Y, x) = \mathrm{S}_0 z$ and $g_1(z, Y, x) = \mathrm{S}_1 z$. The Primitive Person accepted addition, perhaps too readily from a primitive perspective, but balked at doubling. We have no construction of $x \oplus x$ by ramified recurrence, and the recurrence defining exponentiation $\uparrow$ cannot be ramified.

Another ramified recurrence yields the *coercion* $\widehat{\ }\colon \mathsf{W}_1^{\{X\}} \to \mathsf{W}_0$, as follows:

$$\begin{cases} \widehat{\epsilon} = \epsilon \\ \widehat{\mathrm{S}_0 X} = \mathrm{S}_0 \widehat{X} \\ \widehat{\mathrm{S}_1 X} = \mathrm{S}_1 \widehat{X}. \end{cases}$$

Coercion takes a sort 1 string to the identical sort 0 string, but we cannot map in the other direction by a ramified recurrence.

Let SPR, the *sorted primitive recursive functions*, be the smallest set of sorted functions containing the constructors on W_0 and closed under composition and ramified recurrence.

Here is a finer classification. Let SPR_0 be the functions in SPR with no sort 1 variables. If SPR_δ has been defined, let $\mathsf{SPR}_{\delta+1}$ be the smallest set of sorted functions containing SPR_δ and closed under composition and ramified recursion where the given functions are in SPR_δ.

The scheme of ramified recurrence does not confound potentialities with actualities; it is predicative. The Primitive Person would accept each function f in SPR as meaningful, defined on a world of stage f strings closed under the constructors and such that f on stage f strings takes stage 0 values.

4. The Bellantoni-Cook-Leivant theorem

The notion of a machine—Turing machine or register machine—is a syntactical notion. One can encode machines as strings in a straightforward way and then the unary string predicate "x encodes a machine" is in R; in fact, P. Let us assume that such an encoding has been done. Then we say that a unary string predicate p *expresses* a set F of string functions in case for all x, if $x \in p$ then x encodes a machine that computes a function in F and, conversely, for all f in F there is an x in p such that x encodes a machine that computes f. The notion of a construction of a primitive recursive function is also syntactical, and there is a unary string predicate in P that expresses the set of primitive recursive functions. (Observe that we characterize by the predicate p the constructions of primitive recurrence functions, not the primitive recursive functions themselves. The question as to whether two such constructions yield the same function is undecidable.)

It was Ackermann who first produced a recursive function (before the notion had been defined) that is not primitive recursive, by using Cantor's diagonal process. We can enumerate the constructions of primitive recursive functions and we can enumerate the strings; then let f

map the nth string x to the S_0 successor of the value of the function represented by the nth construction on x. More generally, Turing argued as follows. Let F be a set of recursive functions and p a unary predicate in R that expresses F. Then, by diagonalization, F cannot be the set of all recursive functions. The notion of a machine that halts on all inputs is semantic; it requires an infinite $\forall\exists$-search: for all inputs there exists a number such that the computation halts after that number of steps.

If we diagonalize on an enumeration of machines that compute certain polynomial-time functions, the new function need not be polynomial time since the degrees of the polynomials for the machines in the enumeration may be unbounded. This is a first indication that diagonalization, while a central tool in the study of the arithmetical hierarchy, is irrelevant to the polynomial-time hierarchy.

This raises the question as to whether it is possible syntactically to characterize the polynomial-time functions. More precisely, is there a unary predicate in P that expresses the set of polynomial-time functions?

A recurrence is *bounded* in case there is a polynomial in the lengths of the arguments that bounds the length of the value. Cobham [Co] showed that if one starts with a few basic functions, the smallest set of functions containing them and closed under composition and bounded recurrence is the set of polynomial-time functions. Since the notion of a bounded recurrence requires an infinite $\forall\exists$-search, this result does not appear to answer our question. But given a recurrence defining f and a polynomial in the lengths of the variables of f, we can modify the recurrence by testing at each step to see whether the polynomial length bound has been exceeded and, if it has, terminating the recurrence with default value ϵ. Then by Cobham's result every polynomial-time function can be computed by such a modified recurrence (and, in fact, by one in which the termination provision is never invoked). This yields an affirmative answer to our question in a somewhat artificial way.

Bellantoni and Cook [BeCo] [Be] produced an elegant and fundamental syntactical characterization of the polynomial-time functions, inspired by a previous work of Leivant [Le91]. Then Leivant [Le] gave a simpler proof of a somewhat stronger result.

For a sorted function f, let f' be the corresponding string function with sorts ignored, and for a set F of sorted functions let F' be the set of f' for f in F. It is by no means obvious that SPR' is even closed under composition, and no notions concerning polynomials enter into its definition. Here is the stunning result:

Theorem. SPR' *is the set of polynomial-time functions, and* $\mathsf{SPR}_\delta{}'$ *is the set of polynomial-time functions of degree* δ*.*

First we prove the forward direction.

Lemma 1. *Let f be in SPR_δ. Then there is a polynomial τ_f of degree δ and a register machine M such that M computes $f(\vec{X},\vec{x})$ in a number of steps bounded by $\tau_f(|\vec{X}|)$, for all $\vec{X}$ and $\vec{x}$.*

Proof. Let a *move* be either a copy, as in the proof that primitive recursive functions are recursive, or a step. First we prove the weaker result with "steps" replaced by "moves".

The proof is by induction on the construction of f. If f is a constructor let $\tau_f = 1$ (a polynomial in zero variables).

We make the notational convention that if τ_k has been defined for each k in $\vec{k}$ then

$$\hat{\tau}_{\vec{k}} = \sum_{k \in \vec{k}} \tau_k.$$

For a composition $f(\vec{X},\vec{x}) = g\big(\vec{X},\vec{x},\vec{h}(\vec{X},\vec{x})\big)$, let

$$\tau_f(|\vec{X}|) = \tau_g(|\vec{X}|) + \hat{\tau}_{\vec{h}}(|\vec{X}|).$$

Then the canonical copy machine for f computes $f(\vec{X},\vec{x})$ in at most $\tau(|\vec{X}|)$ moves.

For a ramified recursion, for each f in $\vec{f}$ let

$$\tau_f(|Y|,|\vec{X}|) = \hat{\tau}_{\vec{g}}(|\vec{X}|) + \alpha \cdot |Y| \cdot \tau_{\vec{g}_0,\vec{g}_1}(|Y|,|\vec{X}|) + \beta$$

where the constants α and β allow for the moves needed to carry out the loop in the recursion. This proves the weaker result.

One can perfectly well stop here; this is essentially what Leivant does. The motivation for proving the slightly stronger result as stated here is the wish to have the sort 0 strings closed only under the constructors; the Primitive Person would object that they may not be copyable even in principle. There is no problem in copying $\vec{X}$, but we cannot copy $\vec{x}$ in a number of steps independent of $|\vec{x}|$.

There are at most $\tau_f(|\vec{X}|)$ executions of destructor commands in the computation of f, not counting those in the copies, so at most the $\tau_f(|\vec{X}|)$ leftmost bits of the $\vec{x}$ are accessed, again not counting the copies. The polynomial τ_f is known, so before the computation of $f(\vec{X},\vec{x})$ put $\tau_f(|\vec{X}|)$ bits 0 into a counter register and use this to put the leftmost $\tau_f(|\vec{X}|)$ bits of $\rho(x)$, for each x in $\vec{x}$, onto a working register for x. At the end of the computation, if the output register is the working register for some x in $\vec{x}$ put its bits in order to the left of $\rho(x)$. $\square$

The other direction of the theorem, that every polynomial-time function is in SPR', is more surprising. To motivate the proof, we first discuss ramified products of degree δ following [Le, §2.5]. Let

$$\mathrm{Rp}_0(x) = \mathrm{S}_0 x.$$

If $\mathrm{Rp}_\delta(Y_\delta, \ldots, Y_1, x)$ has been defined, we define $\mathrm{Rp}_{\delta+1}$ by ramified recursion on Y as follows:

$$\begin{cases} \quad \mathrm{Rp}_{\delta+1}(\epsilon, Y_\delta, \ldots, Y_1, x) = x \\ \mathrm{Rp}_{\delta+1}(\mathrm{S}_* Y, Y_\delta, \ldots, Y_1, x) = \mathrm{Rp}_\delta\big(Y_\delta, \ldots, Y_1, \mathrm{Rp}_{\delta+1}(Y, Y_\delta, \ldots, Y_1, x)\big). \end{cases}$$

I claim that $\mathrm{Rp}(Y_\delta, \ldots, Y_1, x)$ is S_0 applied $|Y_\delta| \cdots |Y_1|$ times to x. The proof is by induction on δ. The claim holds for $\delta = 0$ since an empty product is 1. Suppose the claim holds for δ and consider $\delta + 1$. Then the claim holds for $Y_{\delta+1} = \epsilon$. Now argue by induction on $|Y_{\delta+1}|$. Using both induction hypotheses, we find that for $c = 0$ or $c = 1$,

$$\mathrm{Rp}_{\delta+1}(\mathrm{S}_c Y_{\delta+1}, Y_\delta, \ldots, Y_1, x)$$

consists of

$$|Y_\delta| \cdots |Y_1| + |Y_{\delta+1}| \cdot |Y_\delta| \cdots |Y_1| = |\mathrm{S}_c Y_{\delta+1}| \cdot |Y_\delta| \cdots |Y_1|$$

bits 0 followed by x, which proves the claim.

For a register machine, we encode the states s by distinct strings $\#s$. Then a configuration is specified by $\mu + 1$ strings $\big(\#s, F(y_1), \ldots, F(y_\mu)\big)$ where μ is the number of registers. Using the Case function, defined by flat recurrence, and composition we construct $\mu + 1$ SPR_0 functions $\vec{\mathrm{S}}$ encoding the step function σ; that is, such that if $\sigma(s, F) = (s^*, F^*)$ then

$$\vec{\mathrm{S}}\big(\#s, F(y_1), \ldots, F(y_\mu)\big) = \big(\#s^*, F^*(y_1), \ldots, F^*(y_\mu)\big).$$

Now we mimic the construction of the Rp_δ, where $\vec{\mathrm{S}}$ plays the role of S_0 and the initial configuration $\big(\#s_1, F_1(y_1), \ldots, F_1(y_\mu)\big)$ plays the role of x.

Let $\vec{\mathrm{S}}_0 \vec{x} = \vec{\mathrm{S}} \vec{x}$. If $\vec{\mathrm{S}}_\delta(Y_\delta, \ldots, Y_1, \vec{x})$ has been defined, we define $\vec{\mathrm{S}}_{\delta+1}$ by ramified recursion as follows:

$$\begin{cases} \quad \vec{\mathrm{S}}_{\delta+1}(\epsilon, Y_\delta, \ldots, Y_1, \vec{x}) = \vec{x} \\ \vec{\mathrm{S}}_{\delta+1}(\mathrm{S}_* Y, Y_\delta, \ldots, Y_1, \vec{x}) = \vec{\mathrm{S}}_\delta\big(Y_\delta, \ldots, Y_1, \vec{\mathrm{S}}_{\delta+1}(Y, Y_\delta, \ldots, Y_1, \vec{x})\big). \end{cases}$$

Lemma 2. *If $\vec{x}$ is the encoded initial configuration, then*

$$\vec{\mathrm{S}}_\delta(Y_\delta, \ldots, Y_1, \vec{x})$$

is the encoded configuration after $|Y_1| \cdots |Y_\delta|$ steps.

Proof. The proof is by induction on δ. The result holds for $\delta = 0$ since an empty product is 1. Suppose the result holds for δ and consider $\delta + 1$. Then the result holds for $Y_{\delta+1} = \epsilon$. Now argue by induction on $|Y_{\delta+1}|$. Using both induction hypotheses, we find that for $c = 0$ or $c = 1$,

$$\vec{\mathrm{S}}_\delta(\mathrm{S}_c Y_{\delta+1}, Y_\delta, \ldots, Y_1, \vec{x})$$

is the encoded configuration after

$$|Y_\delta| \cdots |Y_1| + |Y_{\delta+1}| \cdot |Y_\delta| \cdots |Y_1| = |\mathrm{S}_c Y_{\delta+1}| \cdot |Y_\delta| \cdots |Y_1|$$

steps. □

Lemma 3. *Let f' be a polynomial-time function of degree δ. Then there is an f in SPR_δ such that f' is f with sorts ignored.*

Proof. Let M be a register machine that computes $f'(x_1, \ldots, x_\lambda)$ in at most $\pi(|x_1|, \ldots, |x_\lambda|)$ steps, where π is a polynomial of degree δ. Suppose first that π is a monomial $|x_1|^{\alpha_1} \cdots |x_\lambda|^{\alpha_\lambda}$. The registers for M are the input registers $\rho(x_1)$, …, $\rho(x_\lambda)$ together with certain other registers; let y_1, …, y_μ be x_1, …, x_λ followed by the variables for the other registers. Let

$$f(X_1, \ldots, X_\lambda) = \vec{\mathrm{S}}_\delta(X_1, \ldots, X_1, \ldots, X_\lambda, \ldots, X_\lambda, \widehat{X}_1, \ldots, \widehat{X}_\lambda, \epsilon, \ldots, \epsilon)$$

where there are α_κ occurrences of X_κ, for $1 \le \kappa \le \lambda$, before the coercions $\widehat{X}_1$, …, $\widehat{X}_\lambda$. Then f is in SPR_δ, and by Lemma 2 its value is $f'(X_1, \ldots, X_\lambda)$. Any polynomial π with coefficients in $\mathbb{N}$ is a sum of monomials and the corresponding function is obtained by composition from the functions for the monomials.

This proves the lemma and the theorem. □

5. Cook's theorem

In discussing Cook's theorem, we follow the exposition in Garey and Johnson's book [GaJo], a book that begins with an informative overview of time complexity and contains a wealth of results. Their treatment is based on a kind of Turing machine [GaJo, §2.2] well adapted for studying predicates. It gives the same set P; see [Le, Lemma 3.1], [GaJo, §1.3], and the references [HoUl] [AhHoUl] cited there.

The *satisfiability problem* is to determine whether a conjunction of disjunction of literals (a literal is a Boolean variable or its negation) is satisfied by some truth assignment. This is easily encoded as a string predicate and is readily seen to be in NP. Cook's theorem [Co] states that satisfiability is NP-complete; if it is in P, then $\mathsf{NP} = \mathsf{P}$ and the polynomial-time hierarchy collapses to P. When I presented the beautiful proof following [GaJo, §2.6], Mitsuru Yasuhara observed that the group G_6 of clauses should also contain clauses

$$\{\, \overline{H[i,j]},\ \overline{S[i,j',l]},\ S[i{+}1,j',l]\,\}$$

for $0 \le i < p(n),\ -p(n) \le j < j' \le p(n)+1,\ 0 \le l \le \nu$ to ensure that no symbol l on a square j' other than the square j being scanned changes in the $i+1^{\text{st}}$ step.

See [GaJo] for hundreds of other NP-complete problems, all stemming from Cook's theorem.

Now we resume the formal development of our topic.

6. First-order languages

Our basic reference for mathematical logic is Shoenfield's book [Sh]; see §§2.4–2.5 in connection with this section. Though we will not be taking material directly from Buss's thesis [Bu], we are nonetheless following his lead as we apply mathematical logic to the study of the polynomial-time hierarchy.

For each λ in $\mathbb{N}$, let $\mathcal{F}_\lambda$ be a set of *function symbols of index* λ and let $\mathcal{P}_\lambda$ be a set of *predicate symbols of index* λ, with $\tilde{=} \in \mathcal{P}_2$. Let $\tilde{\neg}$, $\tilde{\vee}$, and $\tilde{\exists}$ be the *logical operators*. A *logical symbol* is a variable, $\tilde{=}$, or a logical operator, and a *nonlogical symbol* is a function symbol or a predicate symbol other than $\tilde{=}$; together they form the *symbols*. A *first-order language* L is a set of symbols containing all of the logical symbols. A *constant* is a function symbol of index 0. Symbols of index 1 and 2 are called *unary* and *binary* respectively.

An *expression* is a finite sequence of symbols, and it is an *expression of* L in case each symbol in it is in L. We use roman letters, possibly with subscript or superscript, to stand for various kinds of expression: u v for expressions, x y z w for variables, f g for function symbols, e for constants, p q for predicate symbols.

Terms are constructed recursively as follows: x is a term; if $\mathrm{f} \in \mathcal{F}_\lambda$ and $\mathrm{u}_1, \ldots, \mathrm{u}_\lambda$ are terms, so is $\mathrm{f}\mathrm{u}_1 \ldots \mathrm{u}_\lambda$. We use a b c d to stand for terms. An *atomic formula* is an expression $\mathrm{pa}_1 \ldots \mathrm{a}_\lambda$ where $\mathrm{p} \in \mathcal{P}_\lambda$. *Formulas* are constructed recursively as follows: atomic formulas are formulas; if u and v are formulas, so are $\tilde{\neg}\mathrm{u}$, $\tilde{\vee}\mathrm{uv}$, and $\tilde{\exists}\mathrm{xu}$. We use A B C D to stand for formulas.

An occurrence of x in u is *bound* in case it is in a part $\tilde{\exists}\mathrm{xA}$ of u; otherwise, it is *free*. Let $\Phi(\mathrm{u})$ be the set of variables occurring free in u. A *designator* is a term or a formula. If u is a designator, the $\mathrm{x}_1, \ldots, \mathrm{x}_\lambda$ are distinct, and no variable occurring in any of $\mathrm{c}_1, \ldots, \mathrm{c}_\lambda$ occurs bound in u, then

$$\mathrm{u}_{\mathrm{x}_1,\ldots,\mathrm{x}_\lambda}[\mathrm{c}_1, \ldots, \mathrm{c}_\lambda]$$

is the designator, called an *instance of* u, obtained by replacing each free occurrence of x_κ in u by c_κ, for $1 \leq \kappa \leq \lambda$.

A formula is *open* in case $\tilde{\exists}$ does not occur in it; it is *closed* in case there is no free occurrence of a variable in it.

A *structure* s for a first-order language L consists of a non-empty set W, called the *universe* of s, and a mapping that assigns to each function symbol f of L of index λ a λ-ary function s(f) on W and to each predicate symbol p of L of index λ a λ-ary predicate s(p) on W, with $\mathrm{s}(\tilde{=}) = \{\, (\mathrm{x}, \mathrm{y}) \in \mathrm{W}^2 : \mathrm{x} = \mathrm{y} \,\}$. We say that L' is an *extension* of L in case every symbol of L is a symbol of L'. Let L' be an extension of L and let s and s' structures for L and L' with the same universe; then s' is an *expansion* of s in case $\mathrm{s}'(\mathrm{u}) = \mathrm{s}(\mathrm{u})$ for every function symbol and predicate symbol u of L.

We shall extend the mapping $\mathrm{u} \mapsto \mathrm{s}(\mathrm{u})$ to all the designators of L in such a way that if u is a term s(u) is a $\Phi(\mathrm{u})$-function and if u is a formula s(u) is a $\Phi(\mathrm{u})$-predicate.

Let s(x) map ξ in $\mathrm{W}^{\{\mathrm{x}\}}$ to ξ_x. Then extend s to all terms of L by recursively defining

$$\mathrm{s}(\mathrm{fa}_1 \ldots \mathrm{a}_\lambda) = \big(\mathrm{s}(\mathrm{f})\big)\big(\mathrm{s}(\mathrm{a}_1), \ldots, \mathrm{s}(\mathrm{a}_\lambda)\big).$$

If C is the atomic formula $\mathrm{pa}_1 \ldots \mathrm{a}_\lambda$ of L, let

$$\mathrm{s}(\mathrm{C}) = \{\, \xi \in \mathrm{W}^{\Phi(\mathrm{C})} : \big((\mathrm{s}(\mathrm{a}_1)(\xi), \ldots, (\mathrm{s}(\mathrm{a}_\lambda)(\xi)\big) \in \mathrm{s}(\mathrm{p}) \,\}.$$

Then extend s to all formulas of L by recursively defining

$$\begin{aligned} \mathrm{s}(\tilde{\neg}\mathrm{A}) &= \complement\mathrm{s}(\mathrm{A}) \\ \mathrm{s}(\tilde{\vee}\mathrm{AB}) &= \mathrm{s}(\mathrm{A}) \cup \mathrm{s}(\mathrm{B}) \\ \mathrm{s}(\tilde{\exists}\mathrm{xA}) &= \Sigma\mathrm{x}\,\mathrm{s}(\mathrm{A}). \end{aligned}$$

For Γ a set of formulas of L, let

$$\mathrm{s}(\Gamma) = \{\, \mathrm{s}(\mathrm{C}) : \mathrm{C} \in \Gamma \,\}.$$

We say that C is *valid in* s, denoted by $\vDash_{\mathrm{s}} \{\mathrm{C}\}$, in case $\mathrm{s}(\mathrm{C}) = \mathrm{W}^{\Phi(\mathrm{C})}$. If C is closed, then $\mathrm{s}(\mathrm{C}) = \mathsf{T}$ or $\mathrm{s}(\mathrm{C}) = \mathsf{F}$. A closed valid formula is said to be *true in* s.

Let $\mathcal{L}$ be the first-order language whose nonlogical symbols are the constants $\tilde{\epsilon}$, $\tilde{0}$, and $\tilde{1}$, the binary function symbols $\tilde{\oplus}$ and $\tilde{\odot}$, and the binary predicate symbol $\tilde{\preceq}$, and let σ be the structure for $\mathcal{L}$ with universe W such that $\sigma(\tilde{\epsilon}) = \epsilon$, $\sigma(\tilde{0}) = 0$, $\sigma(\tilde{1}) = 1$, $\sigma(\tilde{\oplus}) = \oplus$, $\sigma(\tilde{\odot}) = \odot$, and $\sigma(\tilde{\preceq}) = \preceq$. We call σ the *basic structure.*

For each string x, we define a variable-free term $\star x$ of $\mathcal{L}$, called the *name of* x. Let $\star\epsilon = \tilde{\epsilon}$, and for $x \neq \epsilon$ with $|x| = \nu$ let $\star x$ be $\tilde{\oplus} \cdots \tilde{\oplus} \mathrm{e}_1 \ldots \mathrm{e}_\nu$ where there are $\nu - 1$ occurrences of $\tilde{\oplus}$ and e_μ is $\tilde{0}$ or $\tilde{1}$ according as the μth bit of x is 0 or 1, for $1 \le \mu \le \nu$. A term is a *name* in case it is $\star x$ for some string x. We have $\sigma(\star x) = x$ for all strings x.

Having emphasized that the symbols are abstract objects, we now relax the notation to make it more readable. First, we use infix notation for $\tilde{\oplus}$, $\tilde{\odot}$, $\tilde{\preceq}$, $\tilde{=}$, and $\tilde{\vee}$, writing, for example, $\mathrm{A} \mathbin{\tilde{\vee}} \mathrm{B}$ for $\tilde{\vee}\mathrm{AB}$. Second, we introduce defined symbols: $\mathrm{A} \mathbin{\tilde{\&}} \mathrm{B}$ for $\tilde{\neg}(\tilde{\neg}\mathrm{A} \mathbin{\tilde{\vee}} \tilde{\neg}\mathrm{B})$; and $\mathrm{A} \mathbin{\tilde{\rightarrow}} \mathrm{B}$ for $\tilde{\neg}\mathrm{A} \mathbin{\tilde{\vee}} \mathrm{B}$; and $\mathrm{A} \mathbin{\tilde{\leftrightarrow}} \mathrm{B}$ for $(\mathrm{A} \mathbin{\tilde{\rightarrow}} \mathrm{B}) \mathbin{\tilde{\&}} (\mathrm{B} \mathbin{\tilde{\rightarrow}} \mathrm{A})$; and $\tilde{\forall}\mathrm{xA}$ for $\tilde{\neg}\tilde{\exists}\mathrm{x}\tilde{\neg}\mathrm{A}$. Third, we omit the tildes. This means that some notations are used in two different ways; for example, $\oplus$ is used both as a function symbol and as a string function. Occasionally we revert to a more formal notation to avoid ambiguity.

Two formulas A and B are *equivalent in* s in case $\mathrm{A} \leftrightarrow \mathrm{B}$ is valid in s.

7. Bounded formulas

Let x not occur in b. We write

$$\begin{array}{lll} \exists\mathrm{x}{\preceq}\mathrm{b}\,\mathrm{A} & \text{for} & \exists\mathrm{x}(\mathrm{x} \preceq \mathrm{b} \;\&\; \mathrm{A}) \\ \forall\mathrm{x}{\preceq}\mathrm{b}\,\mathrm{A} & \text{for} & \forall\mathrm{x}(\mathrm{x} \preceq \mathrm{b} \rightarrow \mathrm{A}). \end{array}$$

The expressions $\exists\mathrm{x}{\preceq}\mathrm{b}$ and $\forall\mathrm{x}{\preceq}\mathrm{b}$ (which are, respectively, $\exists\mathrm{x}\neg\vee\neg{\preceq}\mathrm{xb}\neg$ and $\neg\exists\mathrm{x}\neg\vee\neg{\preceq}\mathrm{xb}$ in prefix notation) are called *bounded quantifiers* (*existential* and *universal* respectively), and b is the *bound in* the bounded quantifier. A formula is *bounded* in case every occurrence of a quantifier in it is part of a bounded quantifier. We shall be interested exclusively in bounded formulas. Call L a $\preceq$*-language* in case $\preceq$ is a symbol of L and call s a $\preceq$*-structure* in case $\exists x(x \preceq y)$ is valid in s. (This is a very weak condition. It is needed to ensure that, for example, $\exists x{\preceq}b\, A \vee B$, where x does not occur in B, is equivalent in s to $\exists x{\preceq}b(A \vee B)$.)

Call a formula C *adjusted* in case no variable occurs both free and bound and no variable occurs in two different occurrences of quantifiers. A *variant* [Sh, §3.4] of C is a formula obtained from C by changing the bound variables; that is, by repeatedly replacing parts $\exists xB$ by $\exists y B_x[y]$ where y is not free in B. Every formula has an adjusted variant, called its *adjustment*, to which it is equivalent in every structure.

A *bounded prefix* consists of zero or more bounded quantifiers with distinct bound variables; it is *existential* in case they are all bounded existential quantifiers, and similarly for *universal*. In general, a bounded prefix contains both bounded existential and bounded universal quantifiers. A formula C of L is *in bounded prefix form* in case it consists of a bounded prefix followed by an open formula, called the *bounded matrix of* C. Let C be a bounded formula. Construct its *bounded prenex form* by applying the following rules, from left to right and in the order listed, to all parts of the adjustment of C as long as any rule applies. Replace $\neg\exists x{\preceq}bA$ by $\forall x{\preceq}b\neg A$; and $\neg\forall x{\preceq}bA$ by $\exists x{\preceq}b\neg A$; and $\exists x{\preceq}bA \vee B$ by $\exists x{\preceq}b(A \vee B)$; and $A \vee \exists x{\preceq}bB$ by $\exists x{\preceq}b(A \vee B)$; and $\forall x{\preceq}bA \vee B$ by $\forall x{\preceq}b(A \vee B)$; and $A \vee \forall x{\preceq}bB$ by $\forall x{\preceq}b(A \vee B)$. Then the bounded prenex form of C is in bounded prenex form and has the same free variables as C. The bounded prenex form of C is equivalent to C in any $\preceq$-structure.

8. Hierarchies

Let L be a $\preceq$-language. We define a hierarchy of bounded formulas of L. Let $\exists_0(L) = \forall_0(L)$ be the open formulas of L. Then, by recursion, let $\exists_{\nu+1}(L)$ be the set of all formulas of L consisting of a bounded existential prefix followed by a formula in $\forall_\nu(L)$, and let $\forall_{\nu+1}(L)$ be the set of all formulas of L consisting of a bounded universal prefix followed by a formula in $\exists_\nu(L)$. Since empty prefixes are allowed, each $\exists_\nu(L)$ and $\forall_\nu(L)$ is contained in all $\exists_\mu(L)$ and $\forall_\mu(L)$ for $\nu < \mu$.

Now let s be a $\preceq$-structure for L. Let

$$\Sigma_\nu(s) = s\big(\Sigma_\nu(L)\big)$$
$$\Pi_\nu(s) = s\big(\Pi_\nu(L)\big)$$

and let $\Sigma(s)$, the *bounded hierarchy of* s, be the map $\nu \mapsto \Sigma_\nu(s)$. We call $\Sigma(\sigma)$ the *basic hierarchy*.

We say that $\Sigma(s)$ *collapses at* ν in case for every bounded formula C of L we have $\sigma(C) \in \Sigma_\nu(s)$, and we say that $\Sigma(s)$ *collapses* in case it collapses at some ν.

Let s and s$'$ be $\preceq$-structures for L and L$'$ with the same function symbols and the same universe W. We say that $\Sigma(\mathrm{s})$ and $\Sigma(\mathrm{s}')$ are *cofinal* in case there exist α and α' such that $\Sigma_0(\mathrm{s}) \subseteq \Sigma_\alpha(\mathrm{s}')$ and $\Sigma_0(\mathrm{s}') \subseteq \Sigma_{\alpha'}(\mathrm{s})$.

Theorem 1. *Let* s *be a* $\preceq$*-structure for* L. (i) *If* A *and* B *are in* $\exists_\nu(\mathrm{L})$*, then the bounded prenex forms of* A $\vee$ B *and* A & B *are in* $\exists_\nu(\mathrm{L})$. (ii) *If* $\nu > 0$ *and* $\Pi_\nu(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$*, then* $\Sigma(\mathrm{s})$ *collapses at* ν. (iii) *If* $\Sigma_{\nu+1}(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$*, then* $\Sigma(\mathrm{s})$ *collapses at* ν. (iv) *If* $\Sigma(\mathrm{s})$ *and* $\Sigma(\mathrm{s}')$ *are cofinal, then* $\Sigma(\mathrm{s})$ *collapses if and only if* $\Sigma(\mathrm{s}')$ *collapses.*

Proof. We prove (i) by induction on ν. For $\nu = 0$ it is clear. Suppose that $\nu > 0$ and that (i) holds for $\nu - 1$. Then A and B are of the form $\mathrm{Q}_1\mathrm{A}'$ and $\mathrm{Q}_2\mathrm{B}'$ where Q_1 and Q_2 are bounded existential prefixes and A$'$ and B$'$ are in $\forall_\nu(\mathrm{L})$. The bounded prenex forms of A $\vee$ B and A & B are of the form $\mathrm{Q}(\mathrm{A}'' \vee \mathrm{B}'')$ and $\mathrm{Q}(\mathrm{A}''\ \&\ \mathrm{B}'')$ where Q is a bounded existential prefix and A$''$ and B$''$ are variants of A$'$ and B$'$. It is clear that variants of formulas in $\forall_{\nu-1}(\mathrm{L})$ are in $\forall_{\nu-1}(\mathrm{L})$, so the bounded prenex forms of A $\vee$ B and A & B are in $\exists_\nu(\mathrm{L})$ by the induction hypothesis.

To prove (ii), let $\nu > 0$ and $\Pi_\nu(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$. By duality, we have $\Sigma_\nu(\mathrm{s}) \subseteq \Pi_\nu(\mathrm{s})$, so $\Pi_\nu(\mathrm{s}) = \Sigma_\nu(\mathrm{s})$. Hence any formula in $\exists_{\nu+1}(\mathrm{L})$ is equivalent in s to a formula that is of the form QA where Q is a bounded existential prefix and A $\in \exists_\nu(\mathrm{L})$, but this is a formula in $\exists_\nu(\mathrm{L})$. Hence $\Sigma_{\nu+1}(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$ and by duality $\Pi_{\nu+1}(\mathrm{s}) \subseteq \Pi_\nu(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$, so we have $\Sigma_{\nu+1}(\mathrm{s}) = \Pi_{\nu+1}(\mathrm{s}) = \Sigma_\nu(\mathrm{s}) = \Pi_\nu(\mathrm{s})$. By induction, $\Sigma_\mu(\mathrm{s}) = \Sigma_\nu(\mathrm{s})$ for all $\mu \geq \nu$. Since the bounded prenex form of every bounded formula of L is in $\exists_\mu(\mathrm{L})$ for some μ, $\Sigma(\mathrm{s})$ collapses at ν.

To prove (iii), suppose that $\Sigma_{\nu+1}(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$. We always have $\Pi_\nu(\mathrm{s}) \subseteq \Sigma_{\nu+1}(\mathrm{s})$, so $\Pi_\nu(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s})$ and the conclusion follows by (ii).

Finally, let s and s$'$ be cofinal, with α and α' as in the definition. Since their languages have the same function symbols, for all μ

$$\begin{aligned}\Sigma_\mu(\mathrm{s}) &\subseteq \Sigma_{\alpha+\mu}(\mathrm{s}')\\ \Sigma_\mu(\mathrm{s}') &\subseteq \Sigma_{\alpha'+\mu}(\mathrm{s}).\end{aligned}$$

Suppose that $\Sigma(\mathrm{s})$ collapses at ν. Then

$$\Sigma_\mu(\mathrm{s}') \subseteq \Sigma_{\alpha'+\mu}(\mathrm{s}) \subseteq \Sigma_\nu(\mathrm{s}) \subseteq \Sigma_{\alpha+\nu}(\mathrm{s}')$$

for all μ, so $\Sigma(\mathrm{s}')$ collapses at $\alpha + \nu$. Similarly, if $\Sigma(\mathrm{s}')$ collapses at ν' then $\Sigma(\mathrm{s})$ collapses at $\alpha' + \nu'$. $\square$

9. Bounded expansions by definitions

See [Sh, §4.6] in connection with this section. Let s be a $\preceq$-structure for L.

Let L′ be obtained by adding a new predicate symbol p of index λ to L. A *bounded defining axiom for* p is a formula of the form

$$\mathrm{p}(\mathrm{x}_1, \ldots, \mathrm{x}_\lambda) \quad \leftrightarrow \quad \mathrm{D}$$

where D is a bounded formula of L, the $\mathrm{x}_1, \ldots, \mathrm{x}_\lambda$ are distinct, and no variable other than them occurs free in D. We construct an expansion s′ of s, called a *bounded expansion by definition of a function symbol*, by letting $\mathrm{s}'(\mathrm{p}) = \mathrm{s}(\mathrm{D})$. Let A be a formula of L′. The *translation of* A *into* L is the formula, denoted by A^*, obtained by replacing each occurrence of $\mathrm{p}(\mathrm{a}_1, \ldots, \mathrm{a}_\lambda)$ by $\mathrm{D}'_{\mathrm{x}_1,\ldots,\mathrm{x}_\lambda}[\mathrm{a}_1, \ldots, \mathrm{a}_\lambda]$, where D′ is a variant of D in which no variable in one of the a_μ occurs bound. The formulas A and A^* are equivalent in s′, and A^* is a formula of L that is bounded if A is bounded.

Now let L′ be obtained by adjoining to L a new function symbol f of index λ. A *bounded defining axiom for* f is a formula of the form

$$\mathrm{y} = \mathrm{f}(\mathrm{x}_1, \ldots, \mathrm{x}_\lambda) \quad \leftrightarrow \quad \mathrm{y} \preceq \mathrm{b} \quad \& \quad \mathrm{C},$$

where b does not contain y, the formula C is a bounded formula of L, the $\mathrm{x}_1, \ldots, \mathrm{x}_\lambda$, y are distinct, no variable other than them occurs free in $\mathrm{y} \preceq \mathrm{b}$ & C, and the existence and uniqueness conditions

$$\exists \mathrm{y}{\preceq}\mathrm{b}\, \mathrm{C} \quad \text{and} \quad \mathrm{y} \preceq \mathrm{b} \quad \& \quad \mathrm{C} \quad \& \quad \mathrm{y}' \preceq \mathrm{b} \quad \& \quad \mathrm{C}_\mathrm{y}[\mathrm{y}'] \quad \rightarrow \quad \mathrm{y} = \mathrm{y}'$$

are valid in L. We construct an expansion s′ of s, called a *bounded expansion by definition of a function symbol*, by letting $\mathrm{s}'(\mathrm{f})$ be the function whose graph is $\mathrm{s}(\mathrm{y} \preceq \mathrm{b}\ \&\ \mathrm{C})$; the existence and uniqueness conditions ensure that this is well defined. Let A be a formula of s′. The *translation of* A *into* L is the formula, denoted by A^*, obtained as follows. Let B be an atomic formula in A with an occurrence of $\mathrm{f}(\mathrm{a}_1, \ldots, \mathrm{a}_\lambda)$ where f does not occur in any of the a's. (If there is no such occurrence, then f does not occur in A and A^* is A.) Let B_1 be obtained by replacing that occurrence by z, where z does not occur in A, and let B_2 be

$$\exists \mathrm{z}{\preceq}\mathrm{b}_{\mathrm{x}_1,\ldots,\mathrm{x}_\lambda}[\mathrm{a}_1, \ldots, \mathrm{a}_\lambda]\, (\mathrm{C}'_{\mathrm{y},\mathrm{x}_1,\ldots,\mathrm{x}_\lambda}[\mathrm{z}, \mathrm{a}_1, \ldots, \mathrm{a}_\lambda]\ \&\ \mathrm{B}_1),$$

where C′ is a variant of C in which no variable of A occurs bound. Continue in this way, for each atomic formula B occurring in A, until no occurrence of f remains; the result is A^*. The formulas A and A^* are equivalent in s', and A^* is a formula of L that is bounded if A is bounded.

We can iterate these constructions, and an expansion s' of s obtained in this way is called a *bounded expansion by definitions of* s. A bounded formula A of the extended language has a translation A^* into L that is equivalent to A in s', and A^* is a bounded formula of L with the same free variables as A.

We use

$$y = f(x_1, \dots, x_\lambda) \quad \leftrightarrow \quad y \preceq b \quad \& \quad C, \text{ otherwise } y = \epsilon$$

as an abbreviation for

$$y = f(x_1, \dots, x_\lambda) \quad \leftrightarrow \quad y \preceq b \quad \& \quad \big(C \quad \vee \quad (\neg\exists y{\preceq}b\, C \quad \& \quad y = \epsilon)\big).$$

Then the existence condition holds trivially.

We sometimes introduce a function symbol by an explicit definition,

$$f(x_1, \dots, x_\lambda) = c$$

where no variable other than $x_1, \dots, x_\lambda$ occurs in c, in lieu of the defining axiom

$$y = f(x_1, \dots, x_\lambda) \quad \leftrightarrow \quad y \preceq c \quad \& \quad y = c.$$

Then the existence and uniqueness conditions hold trivially.

10. Flat formulas

The material of this section will not be used until much later.

Call a formula A of $\mathcal{L}$ *flat* in case it is bounded and all the bounds in the bounded quantifiers are the same bound, called the *bound of* A (but if A is open, let $\tilde{\epsilon}$ be the bound of A). Notice that if A is flat the only variables occurring in its bound are free in A, since no bound variable can occur in its own bound.

Theorem 2. (i) *Every bounded formula of $\mathcal{L}$ is equivalent in σ to a flat formula.* (ii) *If a bounded formula* A *of $\mathcal{L}$ is equivalent in σ to a formula in $\Sigma_\nu(\mathcal{L})$, then it is equivalent in σ to a flat formula in $\Sigma_\nu(\mathcal{L})$ with the same free variables as* A.

Proof. We construct recursively a formula $A^\flat$ that is flat and equivalent in σ to the bounded formula A. We take variants if need be to ensure that all of the formulas in the recursion are adjusted; $A^\flat$ will be unique modulo adjusted variants. If A is an atomic formula, then $A^\flat$

is A. If A is $\neg$B, then $A^\flat$ is $\neg B^\flat$. If A is $B \vee C$, let b be the bound of $B^\flat$ and c the bound of $C^\flat$. Let $A^\flat$ be obtained by replacing each part $\exists x{\preceq}d\, D$ of $B^\flat \vee C^\flat$ (so that d is b or c) by $\exists x{\preceq}b{\oplus}c(x \preceq d \;\&\; D)$; then $b \oplus c$ is the bound of $A^\flat$. Finally, if A is $\exists x{\preceq}a\, B$, let b be the bound of $B^\flat$. Let

$$b' \quad \text{be} \quad a \oplus b_x[a]$$

and let B' be the formula obtained from $B^\flat$ by replacing each part $\exists y{\preceq}b\, C$ by $\exists y{\preceq}b'(y \preceq b \;\&\; C)$. The formulas

$$\begin{aligned} z_1 \preceq w_1 \quad \& \quad z_2 \preceq w_2 \quad &\to \quad z_1 \oplus z_2 \preceq w_1 \oplus w_2 \\ z_1 \preceq w_1 \quad \& \quad z_2 \preceq w_2 \quad &\to \quad z_1 \odot z_2 \preceq w_1 \odot w_2 \\ z_1 \preceq z_2 \quad \& \quad z_2 \preceq z_3 \quad &\to \quad z_1 \preceq z_3 \end{aligned}$$

are valid in σ. Hence by induction on the number of occurrences of the symbols $\oplus$ and $\odot$ in b,

$$x \preceq a \quad \& \quad y \preceq b \quad \to \quad y \preceq b_x[a]$$

is valid in σ, and so $B^\flat$ and B' are equivalent in σ. Then let $A^\flat$ be $\exists x{\preceq}b'(x \preceq a \;\&\; B')$, so that b' is the bound of $A^\flat$. This completes the construction of $A^\flat$ and proves (i).

Let A be a bounded formula of $\mathcal{L}$ that is equivalent in σ to the formula B in $\Sigma_\nu(\mathcal{L})$. Let $x_1, \ldots, x_\lambda$ be the variables occurring free in B but not in A and let B_1 be $B_{x_1,\ldots,x_\lambda}[\epsilon, \ldots, \epsilon]$. Let $y_1, \ldots, y_\kappa$ be the variables occurring free in A but not in B and let B_2 be

$$B_1 \quad \& \quad y_1 = y_1 \quad \& \quad \cdots \quad \& \quad y_\kappa = y_\kappa.$$

Finally, let B_3 be the bounded prenex form of $B_2^\flat$. Then A is equivalent in σ to B_3, they have the same free variables, and B_3 is a flat formula in $\Sigma_\nu(\mathcal{L})$. $\square$

A flat formula in bounded prenex form is said to be *in flat prenex form*. Let A be a bounded formula of $\mathcal{L}$. The *flat prenex form of* A is the bounded prenex form of $A^\flat$; it is in flat prenex form.

11. Conventions

We associate from left to right, so that $a \oplus b \oplus c$ is $(a \oplus b) \oplus c$. We usually omit $\oplus$, writing ab for $a \oplus b$. We let $\oplus$ bind more tightly than $\odot$, so $ab \odot cd$ is $(a \oplus b) \odot (c \oplus d)$. We use $a \neq b$ to abbreviate $\neg(a = b)$. Brackets as well as parentheses are used for grouping formulas.

Defining axioms are marked *Df*; they are sometimes followed by remarks, with proofs where necessary. A reference to a defining axiom may include the remark following it. We occasionally state that some formulas, marked *Th*, are valid in an expansion by definitions of σ, again with proofs where necessary. We shall simply write σ when some expansion by definitions of σ is meant, and $\vDash$ for validity in some expansion by definitions of σ.

Upper case is used for function symbols and lower case for predicate symbols.

Italic letters, possibly with a digit subscript or primes, stand for strings. Order the italic letters as $A \ldots Za \ldots z$, followed by these with subscript 0, …, 9, followed by all these with one prime, and so forth; call this alphabetical order. If ω is the νth italic letter in alphabetical order, then $\tilde{\omega}$ denotes the νth variable in alphabetical order, though the tilde is omitted in the relaxed notation.

When a constant is introduced by an explicit definition, we also regard it as an abbreviation for the term it is set equal to.

12. Quoting strings

Now we construct some bounded expansions by definitions of σ. First we introduce an expanded alphabet by grouping bits in threes.

Df 1. $\mathtt{0} = 000$.

Df 2. $\mathtt{1} = 001$.

Df 3. $\mathtt{<} = 010$.

Df 4. $\mathtt{>} = 011$.

Df 5. $\mathtt{;} = 100$.

Df 6. $\mathtt{@} = 101$.

Df 7. $\mathtt{:} = 110$.

Df 8. $\mathtt{|} = 111$.

Df 9. $x \approx y \quad \leftrightarrow \quad x \preceq y \quad \& \quad y \preceq x$.

$\vDash\{\star x \approx \star y\}$ if and only if $|x| = |y|$.

Df 10. $x = \text{Triple}\, y \leftrightarrow x \preceq yyy \;\; \& \;\; x \approx yyy \;\; \& \;\; \forall a{\preceq}y \forall c{\preceq}y \exists a'{\preceq}x \exists c'{\preceq}x$ $[a' \approx aaa \;\; \& \;\; c' \approx ccc \;\; \& \;\; (y = a0c \;\; \rightarrow \;\; x = a'\mathtt{0}c') \;\; \& \;\; (y = a1c \;\; \rightarrow \;\; x = a'\mathtt{1}c')]$.

$\sigma(\text{Triple}\, \star y)$ is the string obtained from y by replacing each 0 in y by $\mathtt{0}$ and each 1 by $\mathtt{1}$.

Df 11. $\langle y\rangle = \texttt{<}(\text{Triple } y)\texttt{>}$.

For example, $\vDash\big\{\langle 01101\rangle = \texttt{<01101>}\big\}$.

Df 12. x quotes a string $\;\leftrightarrow\;$ $\exists y{\preceq}x[x = \langle y\rangle]$.

Df 13. threefold x $\;\leftrightarrow\;$ $\exists y{\preceq}x[x \approx yyy]$.

Df 14. a and c are an occurrence of x in y $\;\leftrightarrow\;$ threefold x & threefold a & threefold c & $y = axc$.

Df 15. x occurs in y $\;\leftrightarrow\;$ $\exists a{\preceq}y\exists c{\preceq}y[a$ and c are an occurrence of x in $y]$.

Df 16. x begins with y $\;\leftrightarrow\;$ $\exists z{\preceq}x[x = yz]$.

Df 17. x ends with z $\;\leftrightarrow\;$ $\exists y{\preceq}x[x = yz]$.

Df 18. x quotes strings $\;\leftrightarrow\;$ $x = \epsilon$ $\vee$ $\big[$threefold x & x begins with $\texttt{<}$ & x ends with $\texttt{>}$ & $\forall a{\preceq}x\forall c{\preceq}x\forall s{\preceq}x\big(a$ and c are an occurrence of s in x & $s \approx \texttt{0}$ $\rightarrow$ $[s = \texttt{<} \;\vee\; s = \texttt{0} \;\vee\; s = \texttt{1} \;\vee\; s = \texttt{>}]$ & $[s = \texttt{<} \;\rightarrow\; a = \epsilon \;\vee\; a$ ends with $\texttt{>}]$ & $[s = \texttt{>} \;\rightarrow\; c = \epsilon \;\vee\; c$ begins with $\texttt{<}]\big)\big]$.

$\vDash\big\{\star x$ quotes strings$\big\}$ if and only if $x = \sigma(\langle\star x_1\rangle\ldots\langle\star x_\nu\rangle)$ for unique strings $x_1, \ldots, x_\nu$.

Df 19. z is a list $\;\leftrightarrow\;$ threefold z & z begins with $\texttt{;}$ & z ends with $\texttt{;}$.

Df 20. w is listed in z $\;\leftrightarrow\;$ z is a list & $\texttt{;}w\texttt{;}$ occurs in z & $\neg[\texttt{;}$ occurs in $w]$.

Df 21. $\text{next}(w, w', z)$ $\;\leftrightarrow\;$ z is a list & w is listed in z & w' is listed in z & $\texttt{;}w\texttt{;}w'\texttt{;}$ occurs in z.

13. Turing machines

Following the account in [GaJo, §2.2], we describe the notion of a Turing machine. We modify the exposition to allow for string predicates of any index.

Let Γ be a set of four elements, called *tape symbols*, consisting of the 0-*symbol*, the 1-*symbol*, the *blank symbol*, and the *separator symbol*. Let Q be a finite set of elements called *states*, including the *initial state* and two *halt states*: the *accepting state* and the *rejecting state*. Let Q' be the non-halt states in Q. Then a Turing machine program is completely characterized by a *transition function*

$$\tau\colon\ \mathrm{Q}' \times \Gamma \;\rightarrow\; \mathrm{Q} \times \Gamma \times \{-1, +1\}.$$

We call -1 and $+1$ *moves.*

A *configuration* is a triple $\mathrm{k} = (\mathrm{L}, \mathrm{q}, \mathrm{R})$ where L and R are finite sequences of tape symbols and q is a state. The leftmost tape symbol of R, or the blank symbol if R is empty, is the *scanned symbol* of k.

For any finite sequence S of tape symbols, let Chop S be S with the rightmost tape symbol deleted, let Chip S be S with the leftmost tape symbol deleted, and let Last S be the rightmost tape symbol of S, or the blank symbol if S is empty.

We say that the configuration k' *follows* $\mathrm{k} = (\mathrm{L}, \mathrm{q}, \mathrm{R})$ *by* τ in case the following holds. If q is a halt state, $\mathrm{k}' = \mathrm{k}$. Otherwise, let s be the scanned symbol of k and let $\tau(\mathrm{q}, \mathrm{s}) = (\mathrm{q}', \mathrm{s}', \mathrm{m})$. Then $\mathrm{k}' = (\mathrm{L}', \mathrm{q}', \mathrm{R}')$ where L' and R' are determined as follows:

$$\begin{array}{lll} \mathrm{m} = -1: & \mathrm{L}' = \mathrm{Chop\ L}, & \mathrm{R}' = (\mathrm{Last\ L})\mathrm{s}'(\mathrm{Chip\ R}) \\ \mathrm{m} = +1: & \mathrm{L}' = \mathrm{Ls}', & \mathrm{R}' = \mathrm{Chip\ R}. \end{array}$$

The *length* of a configuration $\mathrm{k} = (\mathrm{L}, \mathrm{q}, \mathrm{R})$ is the number of tape symbols in LR. Notice that the length of the configuration that follows k increases by at most 1.

A *computation* for τ is a finite sequence of configurations in which each successive configuration follows the preceding one by τ. It is a *halting computation* in case a halt state is reached, and an *accepting computation* in case the accepting state is reached.

The *input for* x is the sequence of tape symbols in which each bit 0 of x is replaced by the 0-symbol and each bit 1 by the 1-symbol. The *input for* $x_1, \ldots, x_\lambda$ is the concatenation of the inputs for the x_κ, for $1 \le \kappa \le \lambda$, separated by the separator symbol. The *initial configuration* for $x_1, \ldots, x_\lambda$ is $(\mathrm{L}_0, \mathrm{q}_0, \mathrm{R}_0)$ where L_0 is empty, q_0 is the initial state, and R_0 is the input for $x_1, \ldots, x_\lambda$.

Let us express all of this in σ.

Df 22. s is a tape symbol $\leftrightarrow$ $s = \texttt{<0>}$ $\vee$ $s = \texttt{<1>}$ $\vee$ $s = \texttt{<@>}$ $\vee$ $s = \texttt{<:>}$.

We take $\texttt{<0>}$ as the 0-symbol, $\texttt{<1>}$ as the 1-symbol, $\texttt{<@>}$ as the blank symbol, and $\texttt{<:>}$ as the separator symbol. If $\vDash\{\star s \text{ is a tape symbol}\}$ then $|s| = 9$ by Dfs. 1–7.

Df 23. q is a state $\leftrightarrow$ q quotes a string & q begins with $\texttt{<00}$.

Df 24. $\mathrm{q}_0 = \texttt{<00>}$.

Df 25. $\mathrm{q}_{\mathrm{Y}} = \texttt{<001>}$.

Df 26. $\mathrm{q}_{\mathrm{N}} = \texttt{<000>}$.

Df 27. q is a non-halt state $\quad\leftrightarrow\quad$ q is a state $\;\&\;$ $q \neq \mathrm{q}_{\mathrm{Y}}$ $\;\&\;$ $q \neq \mathrm{q}_{\mathrm{N}}$.

We take q_0 as the initial state, q_{Y} as the accepting state, and q_{N} as the rejecting state.

Df 28. $-1 = \texttt{<10>}$.

Df 29. $+1 = \texttt{<11>}$.

Df 30. m is a move $\quad\leftrightarrow\quad$ $m = -1 \quad\vee\quad m = +1$.

Df 31. t_0 is a transition $\quad\leftrightarrow\quad$ $\exists q{\preceq}x \exists s{\preceq}t_0 \exists q'{\preceq}t_0 \exists s'{\preceq}t_0 \exists m{\preceq}t_0[q$ is a non-halt state $\;\&\;$ s is a tape symbol $\;\&\;$ q' is a state $\;\&\;$ s' is a tape symbol $\;\&\;$ m is a move $\;\&\;$ $t_0 = qsq's'm]$.

If $\vDash\{\star t_0$ is a transition$\}$ then the q, s, q', s', and m are uniquely determined since t_0 is of the form $y_1y_2y_3y_4y_5$ where $\vDash\{\star y_\mu$ quotes a string$\}$ for $1 \leq \mu \leq 5$.

Df 32. t is a transition function $\quad\leftrightarrow\quad$ t is a list $\;\&\;$ $\forall t_0{\preceq}t[t_0$ is listed in $t \;\rightarrow\; t_0$ is a transition$]$ $\;\&\;$ $\forall t_1{\preceq}t \forall t_2{\preceq}t \forall q{\preceq}t \forall s{\preceq}t[t_1$ is listed in t $\;\&\;$ t_2 is listed in t $\;\&\;$ q is a non-halt state $\;\&\;$ s is a tape symbol $\;\&\;$ t_1 begins with $\texttt{;}qs$ $\;\&\;$ t_2 begins with $\texttt{;}qs \;\rightarrow\; t_1 = t_2]$ $\;\&\;$ $\forall q{\preceq}t[q$ is a non-halt state $\;\&\;$ q occurs in $t \;\rightarrow\;$ $\texttt{;}q\texttt{<0>}$ occurs in t $\;\&\;$ $\texttt{;}q\texttt{<1>}$ occurs in t $\;\&\;$ $\texttt{;}q\texttt{<@>}$ occurs in t $\;\&\;$ $\texttt{;}q\texttt{<:>}$ occurs in $t]$ $\;\&\;$ q_0 occurs in t.

Suppose that $\vDash\{\star t$ is a transition function$\}$. Let Q be the set of all q such that $\vDash\{\star q$ occurs in $\star t\}$ together with q_{Y} and q_{N}, and let Γ be the set of four tape symbols. Then t determines the transition function τ defined by letting $\tau(q,s) = (q', s', m)$ where $\vDash\{\star q \star s \star q' \star s' \star m$ is listed in $\star t\}$. This is well-defined because states, tape symbols, and moves are quoted strings beginning in different ways and because q and s determine q', s', and m. Conversely, let Q be a finite set of states containing q_0, q_{Y}, and q_{N} and let τ be a transition function. Let t be the concatenation, separated by ;'s, of all the $qsq's'm$ such that we have $\tau(q,s) = (q', s', m)$. Then $\vDash\{\star t$ is a transition function$\}$. The correspondence is one-to-one modulo the order of listing; chose one such string t and denote it by $\hat{\tau}$.

Df 33. Nine $x = xxxxxxxxx$.

Df 34. ninefold $x \quad\leftrightarrow\quad \exists y{\preceq}x[x \approx \text{Nine } y]$.

Df 35. S consists of tape symbols $\quad\leftrightarrow\quad$ ninefold S $\;\&\;$ $\forall a{\preceq}S \forall b{\preceq}S$ $[$ninefold a $\;\&\;$ $b \approx \texttt{000}$ $\;\&\;$ S begins with $ab \;\rightarrow\; (b = \texttt{<0>} \;\vee\; b = \texttt{<1>} \;\vee\; b = \texttt{<@>} \;\vee\; b = \texttt{<:>})]$.

Df 36. k is a configuration $\leftrightarrow$ $\exists L{\preceq}k\exists q{\preceq}k\exists R{\preceq}k[L$ consists of tape symbols & q is a state & R consists of tape symbols & $k = LqR]$.

Notice that if $\vDash\{\star k$ is a configuration$\}$ then the L, q, and R are uniquely determined since the state q begins in a different way.

Df 37. $y = \text{Chop}\ x$ $\leftrightarrow$ $y \preceq x$ & $\exists c{\preceq}x[c$ quotes a string & $x = yc]$, otherwise $y = \epsilon$.

Df 38. $y = \text{Last}\ x$ $\leftrightarrow$ $y \preceq x$ & y quotes a string & $\exists a{\preceq}x[x = ay]$, otherwise $y = \epsilon$.

Df 39. $y = \text{Chip}\ x$ $\leftrightarrow$ $y \preceq x$ & $\exists a{\preceq}x[a$ quotes a string & $x = ay]$, otherwise $y = \epsilon$.

Df 40. k' follows k by t $\leftrightarrow$ k' is a configuration & k is a configuration & t is a transition function & $\exists L{\preceq}k\exists q{\preceq}k\exists R{\preceq}k\exists L'{\preceq}k'$ $\exists q'{\preceq}k'\exists R'{\preceq}k'\exists s{\preceq}t\exists s'{\preceq}t\exists m{\preceq}t\big[L$ consists of tape symbols & q is a non-halt state & R consists of tape symbols & $k = LqR$ & L' consists of tape symbols & q' is a state & R' consists of tape symbols & $k' = L'q'R'$ & s is a tape symbol & s' is a tape symbol & m is a move & $qsq's'm$ is listed in t & $\big(R$ begins with s $\vee$ $[R = \epsilon$ & $s = \texttt{<@>}]\big)$ & $\big(m = -1$ $\rightarrow$ $L' = \text{Chop}\ L$ & $R' = (\text{Last}\ L)s'(\text{Chip}\ R)\big)$ & $\big(m = +1$ $\rightarrow$ $L' = Ls'$ & $R' = \text{Chip}\ R\big)\big]$.

With the L, R, L', and R' as above, we have $|L'R'| \le |LR| + 9$ by Df. 22, since at most one tape symbol is adjoined.

Df 41. c is a computation for t $\leftrightarrow$ c is a list & t is a transition function & c begins with $;\mathrm{q}_0$ & $\forall k{\preceq}c[k$ is listed in c $\rightarrow$ k is a configuration$]$ & $\forall k{\preceq}c\forall k'{\preceq}c[\text{next}(k, k', c)$ $\rightarrow$ k' follows k by $t]$.

Df 42. c is a halting computation for t on S $\leftrightarrow$ c is a computation for t & S consists of tape symbols & c begins with $;\mathrm{q}_0S;$ & $[\mathrm{q}_{\mathrm{Y}}$ occurs in c $\vee$ q_{N} occurs in $c]$.

Df 43. c is an accepting computation for t on S $\leftrightarrow$ c is a halting computation for t on S & q_{Y} occurs in c.

Df 44. $y = \text{Input}\ x$ $\leftrightarrow$ $y \preceq \text{Nine}\ x$ & $y \approx \text{Nine}\ x$ & $\forall a{\preceq}x\forall c{\preceq}x\exists a'{\preceq}y\exists c'{\preceq}y[a' \approx \text{Nine}\ a$ & $c' \approx \text{Nine}\ c$ & $(x = a0c$ $\rightarrow$ $y = a'\texttt{<0>}c')$ & $(x = a1c$ $\rightarrow$ $y = a'\texttt{<1>}c')]$.

For example, $\vDash\{\text{Input}(01101) = \texttt{<0><1><1><0><1>}\}$.

Df 45. $\text{accepts}(z, t, S)$ $\leftrightarrow$ $\exists c{\preceq}z[c$ is an accepting computation for t on $S]$.

14. Cofinality

A λ-ary predicate p on W is *polynomial time* in case there exist a transition function τ_p and a polynomial π_p in λ variables with coefficients in $\mathbb{N}$ such that for all $x_1, \ldots, x_\lambda$ there is a halting computation on the initial configuration for $x_1, \ldots, x_\lambda$ with at most $\pi(|x_1|, \ldots, |x_\lambda|)$ steps, and it is an accepting computation if and only if $(x_1, \ldots, x_\lambda) \in p$.

For any polynomial π in $|x_1|, \ldots, |x_\lambda|$ with coefficients in $\mathbb{N}$, let $\hat{\pi}$ be the term of $\mathcal{L}$ obtained by replacing each occurrence of $|x_\mu|$ by x_μ for $1 \le \mu \le \lambda$, each occurrence of $+$ by $\oplus$, each occurrence of $\cdot$ by $\odot$, and each coefficient γ by $0 \oplus \cdots \oplus 0$ with γ occurrences of 0 (replacing 0 by ϵ).

Theorem 3. $y \preceq \sigma(\hat{\pi})$ if and only if $|y| \le \pi(|x_1|, \ldots, |x_\lambda|)$.

Proof. By induction on the formation of π. □

Let σ^{p} be the structure for the first-order language $\mathcal{L}^{\mathrm{p}}$ obtained by adjoining to $\mathcal{L}$ a λ-ary predicate symbol p for each λ-ary polynomial-time predicate p, with $\sigma^{\mathrm{p}}(\mathrm{p}) = p$. The hierarchy $\Sigma(\sigma^{\mathrm{p}})$ is the polynomial-time hierarchy, by Wrathall's characterization [Wr] [GaJo, §7.1] [Bu, §1.4] and Theorem 3.

Theorem 4. *The polynomial-time hierarchy and the basic hierarchy are cofinal.*

Proof. The two languages $\mathcal{L}^{\mathrm{p}}$ and $\mathcal{L}$ have the same function symbols, and the basic functions and predicates, as well as the semantic connectives, are all polynomial time, so $\Sigma_0(\sigma) \subseteq \Sigma_0(\sigma^{\mathrm{p}})$. We need only prove that for some α we have $\Sigma_0(\sigma^{\mathrm{p}}) \subseteq \Sigma_\alpha(\sigma)$.

Suppose that $\vDash\big\{\star c$ is a halting computation for $\star t$ on $S\big\}$ and that there are ν steps in the computation (that is, $\nu + 1$ configurations listed in c). I claim that

$$|c| \le (\nu + 1) \cdot |S| + \frac{(\nu + 1) \cdot \nu}{2} \cdot 9 + (\nu + 1) \cdot |t| + (\nu + 2) \cdot 3. \quad (1)$$

By Dfs. 22, 40 and induction, the μ^{th} configuration, for $0 \le \mu \le \nu$, has length at most $|S| + \mu \cdot 9 + |t|$. There are $\nu + 2$ occurrences of ; in c, and $|;| = 3$. This proves the claim.

Let p be a λ-ary polynomial-time predicate. Let π'_p be obtained from the bound in (1) by replacing ν by $\pi(|x_1|, \ldots, |x_\lambda|)$, replacing $|S|$ by $|x_1| + \cdots + |x_\lambda| + (\lambda - 1) \cdot 9$ (the length of the input). and $|t|$ by $|\hat{\tau}|$. Then π'_p is a polynomial in $|x_1|, \ldots, |x_\lambda|$ and

$$p = \sigma\big(\text{accepts}\big(\hat{\pi}'_p, \star\hat{\tau}_p, (\text{Input } x_1)\texttt{<:>}\cdots\texttt{<:>}(\text{Input } x_\lambda)\big)\big).$$

Let A be the translation into $\mathcal{L}$ of accepts$\big(\hat{\pi}'_p, \star\hat{\tau}_p, S\big)$ and let B be the translation into $\mathcal{L}$ of $y = \text{Input } x$, so that $p = \sigma(\text{C})$ where C is

$$\text{A}_S[y_1 \mathtt{<:>} \cdots \mathtt{<:>} y_\lambda] \quad \& \quad \text{B}_{y,x}[y_1, x_1] \quad \& \quad \cdots \quad \& \quad \text{B}_{y,x}[y_\lambda, x_\lambda].$$

The formulas A and B are bounded formulas of $\mathcal{L}$ and so are in $\exists_\alpha(\mathcal{L})$ for some α. (By tracing back through the defining axioms, one finds $\alpha = 4$.) Consequently, C is in $\exists_\alpha(\mathcal{L})$ by part (i) of Theorem 1. Hence $p \in \Sigma_\alpha(\sigma)$. Since p is arbitrary, and the complement of a polynomial-time predicate is polynomial time, we have $\Sigma_0(\sigma^{\mathrm{p}}) \subseteq \Sigma_\alpha(\sigma)$, again by part (i) of Theorem 1. $\square$

Thus we have achieved our first goal. The second goal is to bring nonstandard analysis to bear on the study of the basic hierarchy.