Do We Need Recursion?

Vítězslav Švejdar
Charles University in Prague

Appeared in Igor Sedlár and Martin Blicha eds., The Logica Yearbook 2019, pp. 193–204, College Publications, London, 2020.

Abstract

The operation of primitive recursion, and recursion in a more general sense, is undoubtedly a useful tool. However, we will explain that in two situation where we work with it, namely in the definition of partial recursive functions and in logic when defining the basic syntactic notions, its use can be avoided. We will also explain why one would want to do so.

1 What is recursion, where do we meet it?

Recursion in a narrow sense is an operation with multivariable functions whose arguments are natural numbers. Let $\underline{z}$ denote the k-tuple $z_1, .., z_k$ (where the number k of variables does not have to be indicated), and let $\widehat{f}(x, \underline{z})$ denote the code $\langle f(0, \underline{z}), .., f(x-1, \underline{z})\rangle$ of the sequence $f(0, \underline{z}), .., f(x-1, \underline{z})$ under some suitable coding of finite sequences. If (1), (2) or (3) holds for any choice of arguments:

$$f(0) = a, \qquad f(x+1) = h(f(x), x), \tag{1}$$

$$f(0, \underline{z}) = g(\underline{z}), \qquad f(x+1, \underline{z}) = h(f(x, \underline{z}), x, \underline{z}), \tag{2}$$

$$f(x, \underline{z}) = h(\widehat{f}(x, \underline{z}), \underline{z}), \tag{3}$$

then in the cases (1) and (2) we say that f is derived from a number a and a function h or from two functions g and h by *primitive recursion*, while in the case (3) we say that f is derived from h by *course-of-values* recursion. All functions in (1)–(3) may be partial, i.e., undefined for some arguments. Nevertheless, if h is (both g and h are) total (everywhere defined), then the derived function f must be total. It does not matter that x does not appear in (3) as a variable of h because it can be determined as the length of the sequence encoded in the number $\langle f(0,\underline{z}),..,f(x-1,\underline{z})\rangle$. Course-of-values recursion seems more powerful, but it can be simulated by (derived from) primitive recursion of the form (1) and (2). Therefore, it can be considered a variant of primitive recursion.

Primitive recursion appears in very basic definitions: a function is *partial recursive* if it can be derived from the initial functions using composition, minimization and primitive recursion; it is *primitive recursive* if it can be derived from the same initial functions using composition and primitive recursion only. Since *RE* sets (recursively enumerable sets) are usually defined as the domains of partial recursive functions, the operation of primitive recursion is in fact part of the definition of the arithmetical hierarchy. Primitive recursion is also a useful tool in some proofs. For example, if g is a recursive function with an infinite range, then the equation $f(x) = g(\mu v(g(v) \notin \{f(0),..,f(x-1)\}))$ where $\mu v(..)$ denotes the minimization operation (that is, the search for the first v satisfying the condition in the parentheses) is a derivation by course-of-values recursion of a one-to-one function with the same range. This argument in fact shows that any infinite *RE* set is the range of some one-to-one recursive function.

Recursion in a broader sense is used in *programming languages*: a procedure can be written so that it processes its parameter by calling itself, perhaps several times, with parameters that are simpler in some sense. The parameters do not have to be natural numbers. Also in *logic* we have several definitions that are described as recursive. One example is this: an expression is a term of a language L if it is a variable, or if it is a constant, or if it has the form $F(t_1,..,t_n)$ where $F \in L$ is an n-ary function symbol and $t_1,..,t_n$ are terms. This and other syntactic definitions deal with strings rather that numbers, and they are thus examples of recursion in a broader sense. However, if syntactic objects are identified with natural numbers via some coding of syntax, all these definitions appear to be applications of course-of-values recursion. Then, the set of all terms, the set of all formulas, etc. are primitive recursive sets.

When dealing with metamathematics of Peano arithmetic PA and with incompleteness phenomena, one may need (in fact, does need) arithmetic formulas that define *RE* sets. When dealing with Gödel's second incompleteness theorem, one (of course) needs logical syntax formalized inside Peano arithmetic. In both situations primitive recursion poses a problem because in the arithmetic language (consisting of the binary operation symbols $+$ and $\cdot$, the order symbols $<$ and $\leq$, the constant 0 and the successor function S) there is nothing that would directly correspond to it. One can use existential quantifiers and describe a function that is derived by composition (from functions that can be described), and one can use the least number principle to describe a function that is derived by minimization. However, the dynamic nature of primitive recursion is problematic for the language of a formal theory where we primarily have static descriptions. S. Feferman in his paper [Fef60], which for decades was the most important source of information about Gödel's theorems and about interpretability, introduces the notion of PR-formulas. Its purpose is to have a class of formulas that define and formalize exactly the primitive recursive conditions. Nevertheless, PR-formulas are more an *ad hoc* technical solution than wisely chosen notion that can be further studied. The formalization of syntax itself, and the complexity of the corresponding formulas, cannot be easily learnt from that paper.

While a brief answer to the question whether we need recursion is yes, we will explain that its use in the basic definitions and when formalizing syntax can be replaced by the use of $\mathit{\Delta}_0$ conditions. The class of all $\mathit{\Delta}_0$ conditions is somewhat smaller than the class of all primitive recursive conditions, but it is still quite expressive. It does have a counterpart in the formalized arithmetic, namely the class of Δ_0-formulas. Defining partial recursive functions without recursion may sound paradoxical, and showing that the defined concept remains unchanged does require some effort, but the fact that all *RE* sets are definable in the structure $\mathbb{N}$ of natural numbers then comes practically for free.

All claims and results of this paper are given in more detail in [Šve20]. Practically all ideas concerning formalized syntax are due to Pavel Pudlák: they are either contained in [HP93] or were communicated otherwise. We just add some computations and offer detailed implementation. Defining partial recursive functions without primitive recursion is described in [Odi89].

2 Bounded conditions and bounded formulas

(Multivariable) polynomials in the domain N of natural numbers are functions like $[x, y] \mapsto 2x^2 + 3xy + 1$, obtained from variables and constants by (repeated use of) addition and multiplication. A *bounded condition* (or $\mathit{\Delta}_0$ *condition*) is a condition obtained from equalities of polynomials using Boolean operations and the *bounded quantifiers* $\forall v{\le}f(\underline{x})$, $\exists v{\le}f(\underline{x})$, $\forall v{<}f(\underline{x})$ and $\exists v{<}f(\underline{x})$ where f is a polynomial not dependent on v. Bounded quantifiers have the obvious meaning: $\forall v{\le}f(\underline{x})\,A(v,\underline{x})$ is a shorthand for $\forall v(v \le f(\underline{x}) \Rightarrow A(v,\underline{x}))$ etc. Notice that they interact the expected way with negation: $\neg\forall v{\le}f(\underline{x})\,A(v,\underline{x})$ and $\exists v{\le}f(\underline{x})\neg A(v,\underline{x})$ are equivalent, and similar equivalences hold for the remaining cases. Since one can see a for-loop behind each bounded quantifier in a $\mathit{\Delta}_0$ condition $A(\underline{x})$, it is easy to imagine a program (written in any reasonable programming language) that decides A.

An example of a $\mathit{\Delta}_0$ condition is the divisibility relation: x is a divisor of y (written as $x \mid y$) if $\exists v{\le}y\,(v \cdot x = y)$. Also being a prime is a $\mathit{\Delta}_0$ condition because x is a prime if $x > 1\ \&\ \forall v{<}x\,(v \mid x \Rightarrow v = 1)$. Euclidean division, i.e., the two functions Mod and Div that yield the remainder and the quotient of dividing x by z, have $\mathit{\Delta}_0$ graphs. In the case of the function Div the graph is $\{\,[y, x, z]\,;\ \exists r{<}x\,(x = y \cdot z + r)\ \vee\ (z = 0\ \&\ y = 0)\,\}$, where the purpose of the clause $(z = 0\ \&\ y = 0)$ is to have a total function, which yields some (unimportant) output even when the divisor z is zero. The fact that $y = \mathrm{Mod}(x, z)$ is a $\mathit{\Delta}_0$ condition is proved similarly.

Other examples are the graph $\{\, [y,x]\,;\, y = 2^x \,\}$ of the function $x \mapsto 2^x$ and the range Pwr of this function, i.e., the set $\{1, 2, 4, 8, \dots\}$ of all *powers of two*. Since $\mathrm{Pwr} = \{\, y\,;\, \exists x{<}y\,(y = 2^x)\,\}$, the set (property) Pwr is obtained from the condition $y = 2^x$ using a bounded quantifier, and thus if this condition is Δ_0, then also $\mathrm{Pwr} \in \Delta_0$. This is a sound but misleading observation. The problem about the condition $y = 2^x$ is that it is an equality, but not an equality of polynomials. Thus the fact that it is Δ_0 is a nontrivial result. For a proof see [Ben62] or [Pud83]. Below we will sketch a yet another proof. A direct proof that $\mathrm{Pwr} \in \Delta_0$ is here: x is a power of two if and only if $\forall v{\le}x\,(v \mid x \rightarrow v = 1 \vee 2 \mid v)$.

If r is a power of two, then $\mathrm{Div}(\mathrm{Mod}(u, 2r), r)$ yields the bit in the binary expansion of u that corresponds to the power r. Therefore, using Euclidean division and employing powers of two as pointers, we can speak (in the Δ_0 speach) about binary expansions of numbers. We can, for example, say that the bits corresponding to powers $r_1 < r_2$ are positive while all bits between them are negative. We cannot (yet) say that d is the x-th digit (because that would involve saying that $r = 2^x$). For our proof-sketch that not only $y = 2^x$, but also $y = z^x$ are Δ_0 conditions, consider the following "data structure" consisting of three numbers u, v and w:

$$\begin{array}{rl}
\overbrace{1101}^{x}\;0000000110\;00011\;01 & \quad u\\
\overbrace{110000101001111010011}^{y}\;1011011001\;11011\;11 & \quad v\\
1\;000000000000000000001\;0000000001\;00001\;01 & \quad w\\
\begin{array}{l}\uparrow\\ r_2\end{array}\qquad\qquad\qquad\qquad\begin{array}{l}\uparrow\\ r_1\end{array}\qquad\qquad\qquad\qquad\qquad\qquad &
\end{array} \tag{4}$$

The number w acts as a ruler: the positive bits in its expansion are markers that divide the numbers u and v into items. The separation into items is also indicated by little gaps between digits. The items in u (looking from the right) are the numbers 1, 3, 6 and 13; their binary representations are 1, 11, 110, 1101. The items in v are 3, 27 (whose expansion is 11011), etc. Let $\mathrm{ExpW}(y, x, z, u, v, w)$ be a shorthand for the condition describing the data structure of which (4) is an instance. The condition says "the lowest item in u is 1; the lowest item in v is z and $z > 1$; whenever an item in u is t and the corresponding item in v is e, then the next items in u and v are either $2t$ and e^2 or $2t + 1$ and ze^2; the last (highest) items in u and v are x and y". The condition $\mathrm{ExpW}(y, x, z, u, v, w)$ is Δ_0. If it holds, then u, v and w witness (for $x \neq 0$ and $z > 1$) that $y = z^x$. The condition:

$$\begin{aligned}
&\exists u \exists v \exists w\, \mathrm{ExpW}(y, x, z, u, v, w) \;\vee\\
&\qquad\qquad \vee\; (x = 0 \;\&\; y = 1) \;\vee\; (x \neq 0 \;\&\; z < 2 \;\&\; y = z)
\end{aligned} \tag{5}$$

deals also with the marginal cases where $x = 0$ or $z \le 1$, and it defines the graph of the function $[x, z] \mapsto z^x$. It can be verified that if u, v and w are such that $\mathrm{ExpW}(y, x, z, u, v, w)$, then they do not exceed y^3. This means that the three quantifiers in (5) can be written as $\exists u{\le}y^3$, $\exists v{\le}y^3$ and $\exists w{\le}y^3$. Thus indeed, the condition $y = z^x$ is Δ_0.

The above proof is different from that in [Pud83]. But its main idea (to achieve an efficient data structure by using the recursive conditions $z^{2t} = (z^t)^2$ and $z^{2t+1} = z(z^t)^2$ rather than the condition $z^{t+1} = z(z^t)$) is also due to Pavel Pudlák.

A yet another useful function is the function $x \mapsto \mathrm{NPB}(x)$ that yields the number of positive bits in the binary expansion of a number x. To show that it has a Δ_0 graph, we again design data structure that witnesses that $y = \mathrm{NPB}(x)$. The data structure now consists of a single number w whose binary expansion is seen as a concatenation of (binary expansions of) numbers $S(w, r, i, j)$ that satisfy $S(w, r, i + 1, j) = S(w, r, i, 2j) + S(w, r, i, 2j + 1)$ for every $0 \le i < r$ and $0 \le j < 2^{r-i-1}$, the number r is a power of two not smaller than the total number of bits in x, and $S(w, r, 0, j)$ for each $j < 2^r$ is the j-th digit of x (looking from the right). We call w *summation tree* for x. The numbers $S(w, r, i, j)$ for $i \le r$ and $j < 2^{r-i}$ can be seen as labels of nodes in a binary tree: there are 2^r leaves labeled by the bits in x, each non-leaf is labeled by the sum of the labels of the two children, and the root is labeled by $S(w, r, r, 0)$, which is the result of the whole computation. For example, one can check that the binary expansion of the number $x = 24\,308\,687$ consists of 25 bits. Then the least possible r is 5 and the summation tree w corresponding to x and this r is:

$$\begin{array}{rc}
0\,0\,0\,0\,0\,0\,0\,\overbrace{1\,0\,1\,1\,1\,0\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,1\,1\,1\,0\,0\,1\,1\,1\,1}^{x} & 0 \\
00\;00\;00\;01\;01\;10\;00\;01\;10\;01\;01\;10\;10\;00\;10\;10 & 1 \\
000\;001\;011\;001\;011\;011\;010\;100 & 2 \\
0001\;0100\;0110\;0110 & 3 \\
00101\;01100 & 4 \\
10001 & 5
\end{array}$$

where w is split across six lines (with more significant bits in lower lines). Again, little gaps between digits (and also line breaks) indicate the items $S(w, r, i, j)$ of w. There are $i{+}1$ digits reserved for $S(w, r, i, j)$; the leading zeros in lines $i < r$ are (must be) given, but the leading zero in line 5 is not given because it would be a leading zero in the entire number w. The binary expansion of $S(w, r, r, 0)$ is 10001; indeed, there are 17 positive bits in x. For a full proof that "w is a summation tree for x" is a Δ_0 condition see [Šve20]. It is clear already now that the proof uses the fact that $y = 2^x$ is a Δ_0 condition. It can be proved that for every x there exists a summation tree w such that $w < x^8$, from which it follows that $y = \mathrm{NPB}(x)$ is a Δ_0 condition. The key idea of the proof of this fact is natural: if the positive bits in x are counted not one by one, but by dividing (repeatedly) the binary expansion of x into halves, then the data structure corresponding to the computation is efficient in the sense that it is bounded by a polynomial in x. It should be emphasized that, while we use some ideas from computational complexity and the word 'polynomial' appears several times in this paper, the class of all $\mathit{\Delta}_0$ conditions is different from the classes *NP* and *P* studied in computational complexity.

Knowing that $\mathit{\Delta}_0$ conditions express many useful properties and relations, we are ready to explain their relationship to computability theory and to arithmetic. As already mentioned, partial recursive functions are usually defined as the functions that can be derived from the initial functions by primitive recursion, composition and minimization where the initial functions are $x \mapsto 0$, $x \mapsto x+1$ and $[x_1,..,x_k] \mapsto x_j$ for $1 \leq j \leq k$. It can be shown that if the list of the initial functions is extended by adding addition and multiplication and the function e where $\mathrm{e}(x,y) = 1$ if $x = y$ and $\mathrm{e}(x,y) = 0$ otherwise, then every partial recursive function can be derived without using primitive recursion. This fact is shown in [Odi89] (with citations to Gödel and Kleene). Working with this definition, one can show that the graph (and thus also the domain and the range) of every partial recursive function is a projection of (i.e., a condition obtained by existential quantification from) some $\mathit{\Delta}_0$ relation. On the other hand, it is easy to prove (not employing primitive recursion) that every projection of a $\mathit{\Delta}_0$ relation is the domain of some partial recursive function. From these considerations it follows that the basic notions of computability theory, partial recursive functions and *RE* sets, can be defined without primitive recursion (and actually also without using the functions $[x,z] \mapsto z^x$ and NPB; these are not needed until the next section).

From now on we deal with connections to formal arithmetic. We reserve the letters x, y, etc. for variables in (arithmetic) formulas. Out of many structures, we in fact only need the *standard model* $\mathbb{N} = \langle \mathrm{N}, +, \cdot, 0, \mathrm{S}, \leq, < \rangle$ of Peano arithmetic PA. We use n, k, etc. to denote its elements. We keep in mind that PA is incomplete and has many other models. For $n \in \mathrm{N}$, the n-th *numeral* $\overline{n}$ is the closed term $\mathrm{S}(\mathrm{S}\,..\,(0)\,..)$ containing exactly n occurrences of the successor symbol S. Recall that a formula $\varphi(\underline{x})$ *defines* a set $A \subseteq \mathrm{N}^k$ in $\mathbb{N}$ if the equivalence $A(n_1,..,n_k) \Leftrightarrow \mathbb{N} \models \varphi(\overline{n_1},..,\overline{n_k})$ holds for every k-tuple $[n_1,..,n_k]$. We introduce *bounded quantifiers*: these are quantifiers of the form $\forall v{\leq}t(\underline{x})$, $\exists v{\leq}t(\underline{x})$, $\forall v{<}t(\underline{x})$ and $\exists v{<}t(\underline{x})$ where $t(\underline{x})$ is an arithmetic term not containing v. We define *bounded formulas*, or Δ_0-*formulas*, as formulas in which all quantifiers are bounded. Notice that we use 'Δ' at the formal level and the italic '$\mathit{\Delta}$' at the meta level. The two notions correspond; the sets definable by Δ_0-formulas are exactly the $\mathit{\Delta}_0$ relations, which can be written as follows: $\Delta_0^{\mathbb{N}} = \mathit{\Delta}_0$. This is so because terms in the arithmetic language correspond to multivariable polynomials.

Let $\mathsf{Pwr}(x)$ be the formula $\forall v{\leq}x\,(v \mid x \rightarrow v = \overline{1} \vee \overline{2} \mid v)$ where $v \mid x$ is the formula $\exists u{\leq}v\,(u{\cdot}v = x)$. The formula $\mathsf{Pwr}(x)$ is bounded and defines the set Pwr of all power of two. All our remaining considerations about $\mathit{\Delta}_0$ conditions can be reproduced at the formal level. Thus we can introduce a bounded formula saying that $\mathsf{the\ number}\ y\ \mathsf{is\ the\ number\ of\ positive\ bits\ in}\ x$ and write it as $y = \mathsf{NPB}(x)$, and we can introduce a bounded formula $y = z^x$ that describes exponentiation. We also have bounded formulas $y = \mathsf{Mod}(x, z)$ and $y = \mathsf{Div}(x, z)$ that describe Euclidean division. Notice that we use the single arrow $\rightarrow$ for implication as a symbol in formulas, while the double arrow $\Rightarrow$ was used at the meta level as a shorthand for implication in our speech. We also use the sans-serif font in the shorthands for formulas and also in informal readings of formulas (one should always imagine a formula behind the sans-serif font). We do not systematically invent double symbols: we write the connectives $\&$, $\vee$ and $\neg$, the divisibility symbol $\mid$ and equations like $y = z^x$ the same way at both the formal and the meta level. We do not add new function symbols to the arithmetic language; equations like $y = \mathsf{Mod}(x, z)$ are shorthands for formulas, not equalities of terms.

As already noted, the formula $\mathsf{Pwr}(x)$ defines the set of all powers of two, which means that any n is in Pwr if and only if $\mathbb{N} \models \mathsf{Pwr}(\overline{n})$. The same holds for the remaining formulas. Thus, for example, $\mathbb{N} \models \overline{m} = \mathsf{NPB}(\overline{n})$ if and only if $m = \mathrm{NPB}(n)$, i.e. if and only if the number of positive bits in the binary expansion of n is exactly m. However, our Δ_0-formulas *not only define* the sets that they are supposed to define. They *formalize* the corresponding notions in Peano arithmetic. That is, if they are accepted as definitions of the notions inside PA, then PA can *prove* the expected properties of the notions. "Expected properties" is a vague concept, but in most cases it is clear what to expect. PA can prove that $\mathsf{every\ divisor\ of\ a\ power\ of\ two\ is\ a\ power\ of\ two}$ and that $\mathsf{the\ product\ of\ any\ two\ powers\ of\ two\ is\ again\ a\ power\ of\ two}$. As to the exponential function, $\mathsf{PA} \vdash \forall x \forall y \forall z (z^{x+y} = z^x \cdot z^y)$ and also $\mathsf{PA} \vdash \forall x \forall y \forall z ((z^y)^x = z^{y \cdot x})$. The function NPB has the following summation property: $\mathsf{if}\ s\ \mathsf{is\ a\ power\ of\ two\ greater\ than}\ x,\ \mathsf{then}\ \mathsf{NPB}(y \cdot s + x) = \mathsf{NPB}(y) + \mathsf{NPB}(x)$. It is important to realize that provability (in a theory in general, and in PA in particular) is the same as validity in *all models*. Thus it is good to think of powers of two, the arguments of the exponential function etc. as nonstandard elements of some model of PA. If $\mathcal{M}$ is such a model, it contains nonstandard powers of two, but they behave as expected. Binary expansions can have infinite (nonstandard) length but the positive and negative bits in them can be counted, and $z^{x+y} = z^x \cdot z^y$ holds whether x, y and z are standard or not.

3 Arithmetization of syntax

In this section we deal with syntactic notions formalized in PA. That is, we start with variables and terms and proceed to proofs and provability. We assume that the theory being formalized is PA itself. A straightforward modification of our considerations would make it possible to work in PA with provability in some other theory, say in the Zermelo-Fraenkel set theory ZF.

Since in PA we have numbers and nothing else, we have to specify coding of syntactic objects with numbers. We opt for a coding method that we find natural: characters are identified with their numerical codes, the numerical codes are specified by the *code table* whose size is b, characters in a string w are the digits in the b-ary expansion of w. In more detail, we assume that $\mathsf{b} = 128$ and that the code table is a modified ASCII table. The codes 32–126 are the same as in the ASCII table, the slots 1–31 and 127 (in which the ASCII table has invisible characters) contain characters that are needed in logic but do not occur in the ASCII table (the quantifiers, the set epsilon, . . .), no code is zero. For example, if w is the number $83{\cdot}\mathsf{b}^6 + 40{\cdot}\mathsf{b}^5 + 83{\cdot}\mathsf{b}^4 + 40{\cdot}\mathsf{b}^3 + 48{\cdot}\mathsf{b}^2 + 41{\cdot}\mathsf{b}{+}41$, then w is the code of the string `S(S(0))` because 83, 40, 48 and 41 are the codes (in both the code table and in the ASCII table) assigned to the characters `S`, `(`, `0` and `)`. We write the characters themselves, typeset in the typewriter font, instead of their numerical codes, and we identify strings with their numerical codes. Thus if w is still the same number, we can write $w = \texttt{S(S(0))}$. We omit some of the bars that indicate numerals if there is no danger of misunderstanding.

A number is a *string* if no digit in its b-ary expansion is 0. The *length* $\mathsf{Lh}(w)$ of a string w is the least y such that $w < \mathsf{b}^y$. The x-th symbol $(w)_x$ of a string w (looking from the left) is the number $\mathsf{Div}(\mathsf{Mod}(w, \mathsf{b}^{\mathsf{Lh}(w)-x}), \mathsf{b}^{\mathsf{Lh}(w)-x-1})$ for $x < \mathsf{Lh}(w)$, and it is 0 otherwise. Thus for example, the number 0 is the empty string and its length is 0. The number `0` (notice the typewriter font) is a string having length 1, and we have $(\texttt{0})_0 = \texttt{0}$. We also have $\mathsf{Lh}(\texttt{S(S(0))}) = 7$ and $(\texttt{S(S(0))})_1 = (\texttt{S(S(0))})_3 = 40$ because 40 is the numerical code of the character `(`. Being a string is obviously a Δ_0-formula. Since $y = z^x$ is a Δ_0-formula, also $y = \mathsf{Lh}(w)$ and $y = (w)_x$ are Δ_0-formulas. The *concatenation* $w_1 * w_2$ of w_1 and w_2 is the number $w_1 \cdot \mathsf{b}^{\mathsf{Lh}(w_2)} + w_2$. We omit the symbol $*$ (and write just w_1w_2) if it is clear that we deal with strings.

We need infinitely many variables when writing formulas, and we can assume that there are only countably many of them and that they are indexed by natural numbers. Therefore, we define that a *variable* is a nonempty string consisting of the letter `v` followed by a binary expansion of a number. A binary expansion of a number is a string s consisting of the digits `0` and `1` such that the leftmost digit of s can be `0` only if $\mathsf{Lh}(s) = 1$. Let $\mathsf{Var}(x)$ be the formula $\mathsf{the\ number}$ x $\mathsf{is\ a\ variable}$. Clearly, $\mathsf{Var}(x)$ is a Δ_0-formula. It does not matter (it does not complicate syntax analysis) that the character `0` plays a double role: it appears in indices of variables, and it is also a constant in the arithmetic language. For example, the string version of the formula $\neg\exists v_2(v_2 = 0)$ is `¬∃v10(v10=0)`.

Let $\mathsf{NOcc}(u, w)$ be the *number of occurrences* of the character u in a string w. We again want the function to be total, and thus we assume that it has some (unimportant) values even if u is not a code of a character or if w is not a string. This function can be derived from the function NPB as follows: $y = \mathsf{NOcc}(u, w)$ if and only if there exists a number z not exceeding w such that: the length of the binary expansion of z is $\mathsf{Lh}(w)$, for each x the x-th bit of z is positive if and only if $(w)_x = u$, and $\mathsf{NPB}(z) = y$. Therefore, $y = \mathsf{NOcc}(u, w)$ is a Δ_0-formula. It follows from the summation property of NPB that if w_1 and w_2 are strings, then $\mathsf{NOcc}(u, w_1 * w_2)$ equals $\mathsf{NOcc}(u, w_1) + \mathsf{NOcc}(u, w_2)$.

Using the function NOcc we can define balanced strings, which is the key concept that makes it possible to avoid primitive recursion in logical syntax. It is taken from the part of [HP93] written by Pudlák. We again just add some details and implementations to the treatment of syntax in [HP93]. A string w is *balanced* if its length is at least 2, it contains the same number of left parentheses as right parentheses (in symbols, $\mathsf{NOcc}(\texttt{(}, w) = \mathsf{NOcc}(\texttt{)}, w)$), and whenever u is a nonempty proper initial segment of w, then $\mathsf{NOcc}(\texttt{(}, u) > \mathsf{NOcc}(\texttt{)}, u)$. It is clear that a balanced string must start with `(` and end with `)`. If the concatenation uv of two nonempty strings is balanced, then it follows from the summation property that v contains less left parentheses than right parentheses. *Important property of balanced strings* is this: two balanced substrings u and v of a string w can overlap only if one of them is a substring of the other, and they can start at the same position only if they also end at the same position.

A *quasiterm* is any variable, the (single-element) string `0`, or any string of the form $\texttt{S(}w\texttt{)}$, $\texttt{+(}w\texttt{)}$ or $\texttt{·(}w\texttt{)}$ where $\texttt{(}w\texttt{)}$ is a balanced string. Examples of quasiterms are `+((0))` and `S(()()())`. A quasiterm t is a *term* (which we write as $\mathsf{Term}(t)$) if whenever $\texttt{(}w\texttt{)}$ is a balanced substring of t, then either $\texttt{(}w\texttt{)}$ is immediately preceded by the letter `S` and w is a quasiterm, or it is immediately preceded by `+` or `·` and w has the form $u\texttt{,}v$ (notice the typewriter comma) where u and v are quasiterms. $\mathsf{Term}(t)$ is a Δ_0-formula.

Notice that we write the binary symbols in front of their operands, not between them. To have at least one full proof, let us verify that if t and s are terms and u is the string $\texttt{+(}t\texttt{,}s\texttt{)}$, then u is a term. It is clear that a term contains the same number of left parentheses as right parentheses. Therefore, u contains the same number of left parentheses as right parentheses. Using the summation property of the function NOcc and distinguishing the cases whether the leftmost character of t is `+`, `·`, `v` or `0`, one can conclude that $\texttt{(}t\texttt{,}s\texttt{)}$ is a balanced string. Thus u is a quasiterm. To finish the verification that it is a term, let v be a balanced substring of u. Distinguish the three cases where the leftmost parenthesis of v is inside t, or it is inside s, or it is the leftmost character of $\texttt{(}t\texttt{,}s\texttt{)}$. In the last case it follows from the property of balanced strings that $v = \texttt{(}t\texttt{,}s\texttt{)}$ and thus v is as required in the definition of term (it is preceded by a binary symbol and at the same time it consists of two comma-separated quasiterms surrounded by parentheses). If the leftmost parenthesis of v is inside t, then the property of balanced strings implies that v is a substring of t. Since t is a term, v is as required. If the leftmost parenthesis of v is inside s, then v is as required as well because s is a term. This argument is a proof of a part of claim (b) in the following theorem. The remaining considerations in (b) and (c) are similar. Claim (a) is straightforward, and (d) follows from (a)–(c).

Theorem 1 *(a)* $\mathsf{PA} \vdash \mathsf{Term}(\mathtt{0})\ \&\ \forall v(\mathsf{Var}(v) \to \mathsf{Term}(v))$.

(b) $\mathsf{PA} \vdash$ If t and s are terms, then $\mathtt{S(}t\mathtt{)}$, $\mathtt{+(}t\mathtt{,}s\mathtt{)}$ and $\mathtt{\cdot(}t\mathtt{,}s\mathtt{)}$ are terms.

(c) $\mathsf{PA} \vdash$ Every term u either (i) is the single-element string $\mathtt{0}$, or (ii) is a variable, or (iii) has the form $\mathtt{S(}t\mathtt{)}$ where t is a term, or (iv) has the form $\mathtt{+(}t\mathtt{,}s\mathtt{)}$ or $\mathtt{\cdot(}t\mathtt{,}s\mathtt{)}$ where t and s are terms. The possibilities (i)–(iv) are mutually exclusive and, in (iv), t and s are uniquely determined.

(d) The formulas $\mathsf{Var}(x)$ *and* $\mathsf{Term}(x)$ *define the set of all variables and the set of all terms respectively.*

We see that exactly the same that was said in the previous section about powers of two, about the exponential function and about the function NPB can be said about variables and terms. The formulas $\mathsf{Var}(x)$ and $\mathsf{Term}(x)$ are Δ_0 and they not only define what they are supposed to define, but, if they are accepted as definitions of variables and terms inside PA, then PA can prove the expected properties of those notions. For example, every nonstandard model $\mathcal{M}$ of PA contains infinitely (unboundedly) many terms, and every term w in $\mathcal{M}$ that is not a variable or the constant $\mathtt{0}$ can be uniquely decomposed as $w = \mathtt{+(}t\mathtt{,}s\mathtt{)}$ or $w = \mathtt{\cdot(}t\mathtt{,}s\mathtt{)}$ or $w = \mathtt{S(}t\mathtt{)}$ where t and s are terms. If w is nonstandard and equals $\mathtt{+(}t\mathtt{,}s\mathtt{)}$ or $\mathtt{\cdot(}t\mathtt{,}s\mathtt{)}$, then at least one of t and s must be nonstandard.

The same that we did with variables and terms can be done with the other syntactic notions. Thus we can introduce formulas $\mathsf{Fla}(z)$, $\mathsf{OccT}(v,t)$, $\mathsf{OccF}(v,z)$, $\mathsf{Sent}(z)$, $\mathsf{FreeSub}(v,z,t)$, $\mathsf{SubF}(v,z,t,y)$, and $\mathsf{LogAx}(z)$ that are Δ_0 and express that the string z is a formula, the string t is a term and v is a variable occurring in t, the string z is a formula and v is a variable having free occurrences in z, the string z is a sentence, the string t is a term substitutable for v in z, the string y is the result of substituting a term t for v in a formula z and the string z is a logical axiom respectively. These formulas again have the expected properties. For example, the claim every formula z is either atomic, or it can be uniquely decomposed as $\neg z_1$, $\forall v z_1$, $\exists v z_1$, $(z_1 \to z_2)$, $(z_1 \& z_2)$ or $(z_1 \vee z_2)$ where z_1 and z_2 are formulas and v is a variable can be proved in PA.

4 Conclusions

The use of primitive recursion can sometimes be replaced by the use of $\mathit{\Delta}_0$ conditions and Δ_0-formulas. In particular, the basic syntactic notions in logic and their properties are $\mathit{\Delta}_0$, which is a more accurate result than that they are primitive recursive. This result makes the arithmetization of syntax more natural. Δ_0-formulas that describe terms, formulas etc. are in fact close to computer programs that would decide about a string whether it is a term, a formula, etc.

References

[Ben62] J. H. Bennet. *On Spectra*. Dissertation, Princeton University, Princeton, NJ, 1962.

[Fef60] S. Feferman. Arithmetization of metamathematics in a general setting. *Fundamenta Mathematicae*, 49:35–92, 1960.

[HP93] P. Hájek and P. Pudlák. *Metamathematics of First Order Arithmetic*. Springer, 1993.

[Odi89] P. Odifreddi. *Classical Recursion Theory*. North-Holland, 1989.

[Pud83] P. Pudlák. A definition of exponentiation by a bounded arithmetical formula. *Comm. Math. Univ. Carolinae*, 24(4):667–671, 1983.

[Šve20] V. Švejdar. *Logic: Incompleteness, Complexity, and Necessity*. College Publication, London, 2020. In preparation.