REPORTS ON MATHEMATICAL LOGIC
44 (2009), 19–29

Jerzy MYCKA

A SIMPLE OBSERVATION REGARDING ITERATIONS OF FINITE-VALUED POLYNOMIAL-TIME FUNCTIONS

A b s t r a c t. We present this note to point out that the finite-valued polynomial-time computable functions are closed with respect to iteration. This fact is not a difficult result, however it can be useful in constructions not exceeding the class of polynomial-time computable functions.

1. Introduction

Let us state our problem as follows. We start with some natural function $f : \mathbb{N} \to \mathbb{N}$, which is a computable function. If the function f is, moreover, feasibly computable, is it true that the iteration $F(n, x) = \underbrace{f(f(\ldots(f(x)\ldots))}_{n}$ is feasibly computable too?

Received 29 March 2006

Let us precise the above question. Of course, computable functions are identified in the paper with functions computed by Turing machines. This choice can be modified by use of recursive functions or one of the other standard models of computability (cf. [6]). But a notion of a Turing machine has the advantage by giving clear and simple description of complexity of any computation, which is important for our purpose.

We use the standard definition of Turing machines (for the main concepts of this section good references are e.g. [6, 1]). Let us consider a Turing machine given by the following description. It consists of an infinite tape for storing the input, output, and scratch working, and a finite set of internal states. All elements on a tape are strings. Without any loss of generality, we can choose some alphabet for these strings, the binary alphabet is usually preferred. The machine works in steps. At one step it scans the symbol from the current position of the tape (under the head of the machine), changes this symbol according to the current state of the machine and moves the position of the tape to the left or right with a transformation of state. Some states are distinguished as final, when the machine reaches one of them, then it stops.

Now we can define the class of natural functions computable in Turing sense, derived from the above given description. We assume that we have some fixed coding of numbers on the tape (e.g. binary representations of numbers).

Definition 1. A partial function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be Turing computable iff there exists a deterministic Turing machine M such that (a) M accepts the domain of f and (b) if $\langle x_1, \ldots, x_n \rangle \in \mathrm{dom(f)}$, then the accepting computation writes in the output tape the value $f(x_1, \ldots, x_n)$.

The practical usefulness of such idea of computability can be questioned. It is true that functions like $g(n) = 2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\Big\}n}$ are computable in the above sense, but can we think about $g(100)$ as practically feasible in any sense? For more realistic notion of computability we should direct our attention to feasibility of computational processes. In this case, one cannot ignore restrictions connected with limited, in real world, resources of time and space.

The main tool used in this field is time-complexity (more restrictive than space-complexity). Let us recall the following standard definition.

Definition 2. A Turing machine M has time-complexity given by some natural, computable function $f : \mathbb{N} \to \mathbb{N}$, iff for any input of the length n, the machine M reaches a final state in no more than $f(n)$ steps.

As usual, we will focus on polynomial-time computable functions, as the standard class of physically feasible functions. In this case we define PF as the class of all Turing computable functions with their time complexity realized by any polynomial. We can simply express the above statement that PF is the class of functions computed by Turing machines clocked with polynomials. This class will the main object of our investigation in this paper. Note that for functions in PF the halting problem is decidable.

It is interesting that we have inductive characterizations of the total functions in PF (some of them can be found in [6]). Here we present one of them.

Proposition 3. *The class PF of Turing computable functions in polynomial time is identical with the class inductively defined from the basic functions $Z(n) = 0$ and $S(n) = n + 1$, the projections, basic functions $2n$, $2n+1$, $\lfloor \frac{n}{2} \rfloor$, the characteristic function δ of equality to 0, by the operations of composition, definition by cases, and polynomially bounded primitive recursion.*

We interpret the polynomially bounded primitive recursion in the following way: a function f is defined from functions g, h if there exist polynomials p, q such that

$$f(\vec{x}) = t(\vec{x}, p(|\vec{x}|)),$$

where

$$t(\vec{x}, 0) = g(\vec{x}), t(\vec{x}, y + 1) = h(\vec{x}, y, t(\vec{x}, y))$$

and

$$(\forall y \leq p(|\vec{x}|)) \quad t(\vec{x}, y) \leq q(|\vec{x}|).$$

By $|\vec{x}|$ we mean the length of the vector $\vec{x}$ computed e.g. with respect to the number of digits in its binary representation. Hence, we can interpret $|x_i|$ as $\lfloor \log(x_i + 1) \rfloor$ and the whole vector as the sum of the lengths of all its components.

The intuition behind the above characterization of the total functions in PF is the following: a Turing machine clocked in polynomial time p can write at most $p(|\vec{x}|)$ bits in the output tape for the input $\vec{x}$.

Let us point out that the general schema of recursion does not have the property of creating polynomial-time computable functions from given functions in PF.

Example 4. Let us recall that the multiplication is in PF and moreover, we can use the recursion schema to define $f(x, y) = x^y$:

$$x^0 = 1; x^{y+1} = x^y \times x.$$

Of course, $x^y \notin PF$.

Before considering the next concepts, we should stress that functions in PF can be considered total. The reason is obvious: whenever a function f would be undefined at x we could give to $f(x)$ value 0, because the domain of f is decidable. Hence, for any function f in PF, f can always be considered algorithmically total in PF:

$$\text{new[f](x)} \;=\; \begin{cases} f(x)+1 & \text{if } x \in \text{dom(f)}; \\ 0 & \text{otherwise.} \end{cases}$$

Since for f in PF, dom(f) is a polynomial-time decidable set, the function new[f](x) is in PF. Let us also add that functions in PF can be effectively enumerated (although all total recursive functions cannot be treated in this way). This suggests that, in some sense, polynomial-time computable functions form 'syntactically' defined class of computable functions. Of course, we could extend our presentation for different types of characterizations of polynomial-time computable functions and their relation to other complexity classes (e.g. by safe recursion [2], by real functions [5]) but for the aim of the paper it would not be fruitful.

In this paper we are interested in iteration rather than recursion. However, these two types of operators are closely connected. First, let us give the more formal description of iteration.

Definition 5. A function $F : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined by iteration from a function $f : \mathbb{N} \to \mathbb{N}$ if

$$F(n, x) = f^{(n)}(x),$$

where $f^{(n)}(x)$ denotes the result of n successive applications of f, i.e. $\underbrace{f(f(\ldots(f(x)\ldots))}_{n}$; for $n = 0$ we take as the result x.

We can also use the modification of the above schema to obtain pure iteration, in this case $F(n) = f^{(n)}(0)$.

Let us recall the following consequence of results form [3], [4].

Lemma 6. *If we take for granted the initial functions:* $Z(n) = 0$, $S(n) = n+1$ *and the projections then every function defined with recursion can be defined with iteration instead.*

With some new initial functions we can improve the above result by replacing of iteration by pure iteration, for details see [8].

However, let us recall that recursion can lead beyond PF and that the same situation for polynomial-time computable functions can occur in the case of iteration. Precisely, the iteration of some $f \in PF$ need not to be in PF.

Example 7. Let us start with $g(x) = x^2$. Such function is polynomial-time computable function. Now, let us define f as the iteration of g:

$$f(n, x) = x^{2^n}.$$

Of course, such f in not a polynomial-time computable function.

In this situation we would like to know for what subsets of the class PF (understood as the instance of feasibly computable functions) we have a guarantee that iteration does not lead beyond polynomial-time (i.e. feasibly) computable functions.

2. Iteration and finite-valued functions

In this section we study, as the first step of our analysis, computable functions with finite set of values.

Let us start with the detailed illustration of this problem for functions with two values. We can restrict our considerations, without loss of generality, for the case of $f : \mathbb{N} \to \{0, 1\}$. Now, if such function f is from PF we can ask whether $F(n, x) = f^{(n)}(x)$ is also in PF.

Let us analyze this case. We have such the set A that $f(A) = \{1\}$ and $f(\mathbb{N} - A) = \{0\}$. Now we have the following four cases.

- $0, 1 \in A$: then $f^2(x) = 1$ for all $x \in \mathbb{N}$. In this situation $F(n, x) = 1$ for all $x \in \mathbb{N}, n \geq 2$, therefore $F \in PF$.

- $0, 1 \notin A$: then $f^2(x) = 0$ and the above item can be applied with the replacement of 1 by 0.

- $0 \in A$, $1 \notin A$: then for $x \in A$ we obtain $f^2(x) = f(1) = 0$, and for $x \notin A$ we obtain $f^2(x) = f(0) = 1$. Next $f^3(x) = 1$ for $x \in A$, otherwise $f^3(x) = 0$. Let us observe the consecutive values of $F(n, x)$ for $n \geq 1$ create the sequence of the form $1, 0, 1, 0, \ldots$ for $x \in A$ or $0, 1, 0, 1, \ldots$ for $x \notin A$. Hence, we can present F in the following way for $n \geq 1$:
$$F(n, x) = \begin{cases} 1 - P(n), & f(x) = 1; \\ P(n), & f(x) = 0; \end{cases}$$
where $P(n)$ is the parity function (of course in PF). Again, we obtain $F \in PF$.

- $1 \in A$, $0 \notin A$: then for $x \in A$ we obtain $f^2(x) = f(1) = 1$, and for $x \notin A$ we obtain $f^2(x) = f(0) = 0$. Next $f^3(x) = 1$ for $x \in A$, otherwise $f^3(x) = 0$. Let us observe the consecutive values of $F(n, x)$ for $n \geq 1$ form the sequence $1, 1, 1, \ldots$ for $x \in A$ or $0, 0, \ldots$ for $x \notin A$. Hence, we have simply $n \geq 1$: $F(n, x) = f(x)$. Of course, this gives $F \in PF$.

The above analysis completes the proof of the following lemma.

Lemma 8. *If $f : \mathbb{N} \to \{0, 1\}$ is from PF then $F(n, x) = f^{(n)}(x) \in PF$.*

We can generalize the above lemma for functions with more than two values $f : \mathbb{N} \to \{n_1, \ldots, n_k\}$.

Proposition 9. *If $f : \mathbb{N} \to \{n_1, \ldots, n_k\}$ is from PF then, for $F(n, x) = f^{(n)}(x)$, we have $F \in PF$.*

Proof. Let us proceed with an inductive proof. For $k = 1$ we obtain the constant function. We have additionally presented (as a clarifying illustration) the case with $k = 2$ in the above lemma.

Now let us assume that the theorem holds for some k. We will present the argument that the same statement is always true for $k + 1$. Without

loss of generality, we can consider $f(\mathbb{N}) = \{0, \ldots, k\}$. Let us denote by $A_j, j \leq k$ such sets that $f(A_j) = \{j\}$. It is obvious (because f is total) that $\bigcup_{j=0}^{k} A_j = \mathbb{N}$.

Let us observe that each $i \in \{0, \ldots, k\}$ belongs to some A_j. We should distinguish two cases now.

- For some A_j we have $A_j \cap \{0, \ldots, k\} = \emptyset$. Then $f^2(\mathbb{N}) = \{0, \ldots, k\} - \{j\}$. Now, from the inductive assumption $F_1(n, x) = f^{(2n)}(x)$ is in PF. If we define $F_2(n, x) = f(F_1(n, x)) = f^{(2n+1)}(x)$ then $F_2 \in PF$ too. Finally we can present the following definition to obtain F:

$$F(n, x) = \begin{cases} F_1(\frac{n}{2}, x) & n \text{ is even,} \\ F_2(\frac{n-1}{2}, x) & n \text{ is odd.} \end{cases}$$

  Of course $F(n, x) = f^n(x)$ and F is in PF.

- Every A_j contains exactly one i from $\{0, \ldots, k\}$. Then we have some permutation of values generated by f:

$$\begin{pmatrix} 0 & 1 & \ldots & k-1 & k \\ i_0 & i_1 & \ldots & i_{k-1} & i_k \end{pmatrix}.$$

  It is well known fact that any permutation can be represented as a product of permutation cycles, and this representation is unique (up to the ordering of the cycles). Let us denote these cycles as $C_1, \ldots, C_t$, where t is the number of cycles varying from 1 to k;

$$C_s = \begin{pmatrix} j_0 & j_1 & \cdots & j_{n_s-1} & j_{n_s} \\ i_{j_0} & i_{j_1} & \ldots & i_{j_{n_s-1}} & i_{j_{n_s}} \end{pmatrix}.$$

  Let us observe that for $x \in A_{j_0} \cup \ldots \cup A_{j_{n_s}}$ the iteration $F(n, x)$ will be a periodic function for $n \geq 1$ generated by the cycle C_s with the period equal to n_s. In general, for any x we obtain $F(n, x)$ as a periodic function for $n \geq 1$ too, but with the period $n_1 \times \ldots \times n_t$. Hence, we have the following form of F:

$$F(n, x) = \begin{cases} x & n = 0, \\ f^{(n_1 \times \ldots \times n_t)}(x) & n \mod (n_1 \times \ldots \times n_t) = 0, \\ f^{(k)}(x) & n \mod (n_1 \times \ldots \times n_t) = k. \end{cases}$$

The above conditional function is defined by use only a constant number of compositions of f, hence it is in PF.

In such a way we obtain that for every polynomial-time computable function with a finite set of values, the iteration of one-argument function is still polynomial-time computable. □

The next step is needed to justify the above proposition in the case of functions with any number of arguments. So, let us concentrate our efforts on the following type of functions $f : \mathbb{N}^n \to \mathbb{N}^n$, $f = (f_1, \ldots, f_n)$. In this multi-argument case iteration can be given as follows

$$F(k, \vec{x}) = (f_1(F(k-1, \vec{x})), \ldots, f_n(F(k-1, \vec{x}))).$$

Proposition 10. *Let $f : \mathbb{N}^n \to \mathbb{N}^n$ be a function from PF with a finite set of n-tuples which can be obtained as values. Then, if $F(n, \vec{x}) = f^{(n)}(\vec{x})$, the function $F \in PF$.*

Proof. Of course, this case can be reduced to the previous one. It is enough to use one of standard codings $\bar{x} = \langle x_1, \ldots, x_n \rangle$ and decodings $x_i = (\bar{x})_i, 1 \leq i \leq n$. We should guarantee that these functions are from PF. But this can simply obtained, for example, if we use Turing machines to write n binary representations of $x_1, \ldots, x_n$ as the one number constructed by interlacing of digits of numbers extended by zeroes to have the same length of all binary representations. This process and its inversion are obviously polynomial-time computable for fixed n.

Now, it is sufficient to define $g : \mathbb{N} \to \mathbb{N}$ in the following way $g(\bar{x}) = \langle f(x_1, \ldots, x_n) \rangle$, where $\bar{x} = \langle x_1, \ldots, x_n \rangle$. As a consequence of this definition $g(x) = \langle f((x)_1, \ldots, (x)_n) \rangle$, which means that for $f \in PF$ also g is in PF.

To finish the proof we need to write the below obvious equations.

$$\langle F(k, \vec{x}) \rangle = g^{(k)}(\langle \vec{x} \rangle),$$

$$F(k, \vec{x}) = ((g^{(k)}(\langle \vec{x} \rangle))_1, \ldots, (g^{(k)}(\langle \vec{x} \rangle))_n).$$

Because the right sides of the above equalities are in PF, hence the iteration F is in PF too. □

3. Further remarks about finite-valued computable functions

Because iteration is closely related to primitive recursion we can ask the following question: if we use g, h from PF with a finite set of values to define a new function f by primitive recursion

$$f(\vec{x}, 0) = g(\vec{x}), \quad f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y)),$$

is f in PF too?

We can start our analysis recalling the standard method of a translation between recursion and iteration. It is sufficient to observe that the following transition realizes one steps of recursion: $(z, \vec{x}, y) \longrightarrow (h(\vec{x}, y, z), \vec{x}, y + 1)$. Now by iteration which starts with $z = g(\vec{x}), y = 0$ we obtain the consecutive values of $f(\vec{x}, y)$. However, even for finite-valued g and h this simulation cannot be regarded as reducing finite-valued recursion to finite-valued iteration, with respect to the unbounded element $y + 1$. Hence, this problem needs different methods to find a solution.

To be sure of a relevance of the above considered topics we will give some examples of finite-valued recursive functions, which are not in PF. It would be nice to point out Ackermann's function, which is a classical example of functions beyond PF (as we know beyond the class of primitive recursive functions), but this functions does not have a finite set of values. By the way, the problem of characteristic function for the range of Ackermann's functions was solved (see [9]) with somehow intriguing result that this characteristic function is primitive recursive.

But of course, we can use characteristic functions of languages which are beyond of P. In this context we can choose languages in sufficiently high class of complexity (cf. [1]). Let us recall that this argument is correct only assuming (not yet proved, but strongly supported) inequalities: $P \neq NP$ or $P \neq PSPACE$.

We start with the second mentioned class. For example, the language of quantified boolean formulas can be taken into consideration. In computational complexity theory, the quantified boolean formula problem (QBF) is a generalization of the boolean satisfiability problem. In this case existential or universal quantifiers can be applied to each boolean variable. We distinguish the class of these formulas, which can be satisfied (i.e. which can obtain the value TRUE). For example, the following formula is an instance of QBF: $\forall x(x \vee \neg x)$. QBF is the complete problem for $PSPACE$, the

class of problems solvable by a deterministic (or nondeterministic) Turing machine in polynomial space and unlimited time. With any indexing of the set of quantified boolean formulas and with values 1 for satisfiable and 0 for unsatisfiable, we have a desirable example of a finite-valued function not in P: $f(x) = 1$ iff x is an index of some QBF satisfied formula; 0 otherwise.

Now let us go up. We can define the class of extended regular expressions, which denote some languages over the binary alphabet. Let these expressions can be built like in [7] from the alphabet $\{0, 1, \emptyset, \cdot, \cup, ^\star, \neg\}$. The semantic is given by the obvious equations $L(\emptyset) = \emptyset, L(0) = \{0\}, L(1) = \{1\}$. Now we have the following operations $L(\alpha \cdot \beta) = \{xy : x \in L(\alpha), y \in L(\beta)\}$; $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$; $L(\alpha^\star) = \{x \in L(\alpha) \cdot \ldots \cdot L(\alpha)\}$. Finally, we put $L(\neg\alpha) = \{0, 1\}^\star - L(\alpha)$. Two expressions α, β are equivalent if their languages are identical: $L(\alpha) = L(\beta)$. With the usual numbering of such expressions, we can define the characteristic function of the above equivalence problem. But, it is well known fact that this problem is not even in the class of elementary functions (so, also not in PF).

In the end let us discuss the simple NP-complete problem. The knapsack problem is defined in the following way: we have a finite set of natural numbers $S = \{k_1, \ldots, k_n\}$ and some number $k \in \mathbb{N}$. We say that the problem has a positive solution if there exists some index set $I \subset \{0, \ldots, n\}$ such that $\sum_{i \in I} k_i = k$. Let us code the set S and k into one number $p = \langle k, k_1, \ldots, k_n \rangle$. Then the characteristic function can be defined by the following equations:

$$f(\langle k, k \rangle) = 1; f(\langle k, m \rangle) = 0 \text{ for } k \neq m;$$

$$f(\langle k, k_1, \ldots, k_n \rangle) =$$

$$= \sum_{i=1}^{n} [f(\langle k, k_1, \ldots, k_{i-1}, k_{i+1} \ldots, k_n \rangle) + f(\langle k - k_i, k_1, \ldots, k_{i-1}, k_{i+1} \ldots, k_n \rangle)],$$

where the above sum is taken modulo 2.

Here we have a kind of a recursive definition with the functions of the right side with values in $\{0, 1\}$, but the function f as NP-complete is not in PF (as usual with the assumption $P \neq NP$).

Now let present a few question connected with the result of this paper and opened for the further research.

- Can we establish the probability that for randomly chosen function f from PF its iterations $F(n, x) = f^{(n)}(x)$ is in PF?

- We know that if p is a polynomial and g is periodic then $f(x) = p(x) + g(x)$ is in PF. Is the converse true?
- Moreover, every iteration of finite-valued recursive function is periodic recursive function. Is it possible to present the converse statement: every periodic recursive function is iteration of some finite-valued recursive function?

If the above two questions have the positive answer then we could formulate the following characteristic of polynomial-time computable functions: every $f \in PF$ is a sum of the some polynomial and some iteration of finite-valued function from PF.

References

[1] J. L. Balcázar, J. Díaz, and J. Gabarró, Structural Complexity I, Springer-Verlag, Second Edition, 1995.

[2] S. Bellantoni, S. Cook, *A new recursion-theoretic characterization of the polytime functions*, Computational Complexity, **2**, 2 (1992), pp. 97–110.

[3] M.D. Gladstone, *A reduction of a recursive scheme*, J. Symb. Logic, **32** (1967), pp. 505–508.

[4] M.D. Gladstone, *Simplification of the recursion scheme*, J. Symb. Logic, **36** (1971), pp. 653–665.

[5] J. Mycka, J. F. Costa, *The $P \neq NP$ conjecture in the context of real and complex analysis*, Journal of Complexity, **22** (2006), pp. 287–303.

[6] P. Odifreddi, Classical Recursion Theory, Elsevier, Vol. I, 1992, Vol. II, 1999.

[7] C. Papadimitriou, Computational Complexity, Addison-Wesley 1994.

[8] R.M. Robinson, *Primitive recursive functions*, Bull. Am. Math. Soc., **53** (1947), pp. 925–942.

[9] S.M. Tataram, *Ackermann-Péter's function has primitive recursive graph and range*, Found. Control Engrg., **9**, 4 (1984), pp. 177–180.

Institute of Mathematics,
University of Maria Curie-Sklodowska, Lublin, Poland,
jerzm@hektor.umcs.lublin.pl