SEMINARE
vol. 41 * 2020, no. 4, p. 45-57
DOI: http://doi.org/10.21852/sem.2020.4.04

Fr. Janusz Mączka SDB[1]
The Pontifical University of John Paul II in Cracow
ORCID ID: 0000-0001-8963-9529
Fr. Adam Olszewski[2]
The Pontifical University of John Paul II in Cracow
ORCID ID: 0000-0003-3069-7518

THE QUESTION OF THE BOUNDARIES OF COMPUTER SCIENCE

Abstract

A thesis defining the boundaries of computer science is first formulated in relation to a certain work of Stuart Shapiro. It is argued that these boundaries are determined by the properties of the mind in the real world. It is the Church thesis that determines and specifies these boundaries. Next, some issues concerning the understanding of the Church thesis will be considered such as the division of its formulations and versions. Finally, the relations among the classes of the recursive functions, algorithms, machines and computer programs will be discussed as the main theme of the article. Comments will also be made in the text on the understanding of the term 'effectively'.

Keywords: boundaries, computer science, Church's thesis, effectively, machines, mind

PROBLEM GRANIC INFORMATYKI

Abstrakt

W pracy postawiona zostaje teza dotycząca granic informatyki w nawiązaniu do pewnej pracy Stuarta C. Shapiro. Podaje się argument za tym, że owe granice są określone przez własności umysłu w świecie rzeczywistym. Teza Churcha jest właśnie tym czynnikiem, który wyznacza i specyfikuje owe granice. Potem rozważane są pewne zagadnienia związane z rozumieniem tezy Churcha jak: podział jej sformułowań na warianty i wersje. Następnie, co jest głównym tematem pracy, przedyskutowane są relacje zachodzące pomiędzy klasami funkcji rekurencyjnych, algorytmów, maszyn i programów komputerowych. W tym kontekście podany jest także komentarz na temat rozumienia terminu 'effectively'.

Słowa kluczowe: granice, informatyka, teza Churcha, efektywnie, maszyny, umysł

[1] Janusz Mączka SDB a Professor of the Pontifical University of John Paul II in Cracow. E-mail:januszm@sdb.krakow.pl.

[2] Adam Olszewski, a Professor of the Pontifical University of John Paul II in Cracow; Lectures on Logic and Philosophy. E-mail:atolszad@syf-kr.edu.pl.

¿Todo es un procedimiento?
Anonymous

1. A clarification concerning the title

The question of the limits of a science stems from the more general question connected to the limits of science as a whole. For centuries, philosophers, as well as people of science in general, have tried to make a precise distinction between science and non-science, and also between science and pseudo-science. In both cases, it has been a question of establishing their boundary. The concept of a boundary is an intuitive one since it is used in our everyday lives. It also occurs in mathematics, mainly in two versions which strictly encode two common intuitions related to the notion. The first intuition is expressed as an approximation, in the form of a sequence limit as it is applied in mathematical analysis. The second concept of a boundary is the division of an area in terms of a border as it is utilized in geometry (and topology). In this paper, I will use the intuitive properties of the border as it is used in geometry, in that it separates adjacent areas which are always present in some nonempty space.[3] Therefore, for our purposes, it is crucial to determine the indicated space and to this end two methods present themselves: the first - the syntactic - takes language as its starting point, and distinguishes the language of the discipline within it; the second - the semantic - takes a class of non-linguistic objects as its operating space, and distinguishes it from a subset of objects that are of interest to the specialists of the discipline in question. The latter seems to be the more natural and appropriate of the two, and hence it will be adopted in this paper.

It should also be noted that the concept of a boundary, even in its popular understanding, is not of a relative or subjective nature, but of a normative-objective nature. Our considerations will be directed towards the field of computer science and a more precise reading of the title would be that we intend to examine the boundaries of this subject. The problem which emerges from our deliberations is both interesting and important. While only one universe or domain is usually considered in a theory, the matter is somewhat more complex in computer science. However, this is perhaps far from unusual, given that logic, for example, considers and accounts theoretically for so-called many-sorted logics, i.e., systems with many or multisortal domains.[4] It is worth noting at this point that the question of the boundaries of computer science

[3] In topology, the *explicatum* of the intuitive concept of a boundary would be the edge of a set which closes an open set. A topological explication of the concept leads to surprising cases, such as clopen sets, where the boundary is an empty set. In relation to science, Marciszewski distinguishes in the English sense between a *limit*, which cannot be exceeded, and a *frontier*, which can. This distinction is an important one and it should be emphasized that this paper is concerned with establishing the limits rather than the frontiers of computer science.

[4] More precisely, these are the so-called *sorts* created as a result of dividing one domain into separate subsets (the so-called *many-sorted logics*). Of course, it serves merely as an analogy for the many domains that we attribute to computer science. This course of action may be employed the case of computer science in an effective manner, albeit a slightly artificial one.

is a unique one, one which differs from the traditional question of the material object, the formal object, the formal object *quo* and *quod* of this science, and it is not our intention to seek answers to this second question. This reservation is important, given that Turner & Angius, for example, mention more classes of objects in their work that computer science is concerned with than this paper does.

2. Shapiro's conception and some important reservations

Shapiro attempts to characterize the domain of computer science by establishing a set of all of its procedures, where a procedure as defined by the Merriam-Webster dictionary (*Webster* 1993), is "a particular way of doing or of going about the accomplishment of something." For Shapiro, procedures "are not natural objects," but rather "natural phenomena that may be, and are, objectively measured, principally in terms of the amount of time they take (for those procedures that terminate), and in terms of the amount of resources they require"[5] (Shapiro 2014, 2). An implication of this understanding is that he believes computer science to be a natural science. Such an understanding of the domain of computer science is not common. However, given its imprecise nature it requires further analysis. The definition of a procedure provided by Shapiro includes the term "a particular way of doing," which establishes the proximal type for procedures (*genus proximum*). It should be noted that the way of doing things is generally different from the procedures themselves. What can be empirically observed and investigated are procedures, rather than ways of doing, and while the former are natural phenomena, the latter are not. Proceedings are more of a realization or an exemplification of a way of doing things. This distinction is linked to traditional philosophical problems - universals vs. individuals, or types vs. tokens. Instead, the ways of doing things (procedures) are abstract objects rather than the natural phenomena that Shapiro desires. Of course, there may be doubts whether such an interpretation is appropriate, and whether the interpretation of the way of doing something as a set of physical acts/actions that result in a specific course of a certain physical process should be disallowed. As a result, we have formulated the outline of an argument against the latter interpretation.[6]

- Let us consider *a way of doing something to accomplish something*; let us call it W1, and its repetition W1';
- A second *way of doing something to accomplish something* is required for our purposes; let us call it W2;
- The three ways, W1,W2,W1' are different from one another since, as natural phenomena or physical objects, they (by definition) have different spatial and/or temporal parameters;
- Additionally, let us assume that W1, W1', and W2 add natural numbers;

[5] All citations are from Shapiro's original article in English.

[6] Discussing this issue in more detail, and similar issues from the rest of the work, would require much more space than the authors have at their disposal and they are aware that this issue requires further investigation.

- In order to state on the basis of the empirical observations of W1, W1', and W2, that they are somehow 'identical', it is necessary to refer to something that instantiates itself primarily in W1, W1', and even W2, which is common to them all and has at least a mental character, or is some kind of an abstract object.

For a better understanding of the outline presented above, one may say that "any characterization of computer science must contain some basis which allows the identification of different implementations of a given algorithm, and any such basis requires an abstraction beyond the purely empirical."[7] The above argument should show that the experimental nature of computer science is doubtful if we were to deduce it only from the ways of doing things (as natural phenomena). The argument contradicts Shapiro's claim that the subject of computer science are procedures, understood as a subclass of 'ways of doing' (the third premise of the argument is false). On the other hand, the partially experimental character of computer science, as well as its extent, which we will present further, can be deduced from research on those doings (proceedings) that can be investigated intersubjectively, since they are of an empirical nature.

3. A description of the relations between some of the objects of computer science

A more detailed description of this matter allows us to draw interesting conclusions regarding the boundaries of computer science. Let us consider at least five types of objects from an ontological point of view, all of which are certainly related to computer science.

A. Functions (effectively) calculable (Func);
B. Algorithms (ways of doing) (Alg);
C. Computer programs (in a certain aspect, ways of doing) (Prog);
D. Realizations (doings) (Real);
E. Machines (Machine).[8]

The ways in which these objects exist, or their ontological status, differs. Let us first note that, generally speaking, the existence and nature of such objects may be a) Abstract; b) Mental; c) Physical.

Traditionally, and for our purposes, three main views are typically discerned on the basis of this list which supply the indicated ontic status of objects:[9] platonism - logicism (a.); conceptualism - intuitionism (b.); nominalism - formalism (c.).[10]

[7] We are grateful to the careful reviewers of this paper for this observation.

[8] We understand the term 'Machine' here very broadly and we include theoretical machines, which are actually programs (sometimes program-scripts), but also all devices called computers, which allow one to execute program-scripts. This is the so-called *hardware*.

[9] First, the traditional name was given, followed by a more contemporary name for these philosophical views.

[10] Platonism, for example, does not exclude the existence of b. or c. objects, nor does conceptualism exclude c. objects, but rather they do not necessarily require their existence.

From our list, theoretically speaking, the objects (Func) and (Alg) could be assigned to types a., b. or c. However, the ontological status of the computer programs (Prog), (Real), and (Machine) is more limited. There are some differences among all the indicated classes, e.g., (Prog) contains only artifacts, similarly to (Real) and (Machine) - which consist *almost*[11] exclusively of artifacts, unlike the (Func) and (Alg) classes. Secondly, the objects in the classes (Func) and (Alg) correspond to *intuitive concepts*,[12] unlike the other classes. (Eden 2007, 4-5) distinguishes two senses of the term program: a program-script, understood as a "well-formed sequence of symbols in a programming language," and a program-process - a "process of computation generated by 'executing' a particular program-script" (otherwise referred to as a thread, task, or bot).

Do these two different meanings point to two different types of objects that are defined as programs? It is unclear how this distinction relates to the characteristics of the three paradigms of computer science: rationalist (RAT), technocratic (TEC), and scientific (SCI) (cf. Eden 2007) These refer, among other things, to the three ways of understanding programs as abstract objects, syntactic objects, and mental objects. According to our classification, Eden's program-scripts belong to the class (Prog) and are of the type (c.).[13] The above considerations show that the philosophical standpoint adopted by the researcher is important when considering the ontology of computer science. The richer the ontology assumed at the outset, the more that can be said about the entities that belong to the ontology of computer science. In this paper, therefore, precisely such a rich Platonic ontology is assumed. For example, naturalism, in its extremely ontological version, will have difficulty describing and explaining certain phenomena.[14] In the case of naturalism or any other form of reductionism, abstract beings such as functions or algorithms will have to be imitated by means of the beings which the ontology permits. The philosophical stance adopted here allows for the distinction between the three 'layers' of reality already indicated: physical reality, mental (conceptual) reality, and abstract (ideal) reality. These object layers differ significantly in regard to the properties of the objects they contain. Functions defined over natural numbers constitute a set which, as we know from set theoretical considerations, is of size the power of the continuum. In it, we distinguish a subset of calculable functions, one which is infinite and countable due to the finiteness of their descriptions (cf. Kielkopf 1978). The second issue is that the class of all algorithms is also very extensive, as shown in the figure below derived from the work of Burgin, where the

[11] This means that they also contain natural objects, such as some physical processes that can be said to calculate. However, this is a matter that goes beyond the scope of this work.

[12] This term is partially technical because it was used in the original formulation of Church's Thesis.

[13] The literature often speaks of the dual nature of computer programs - of a concrete and an abstract ontology. Parsons allows such objects to exist, calling them quasi-concrete, and gives sentence-types as an example. Parsons (2008, 33-39).

[14] For reasons of space, we limit ourselves here to merely flagging this potential problem rather than exploring it n more depth

different types of machines/automatons corresponding to the sets of algorithms are shown (Burgin 2005, 250).

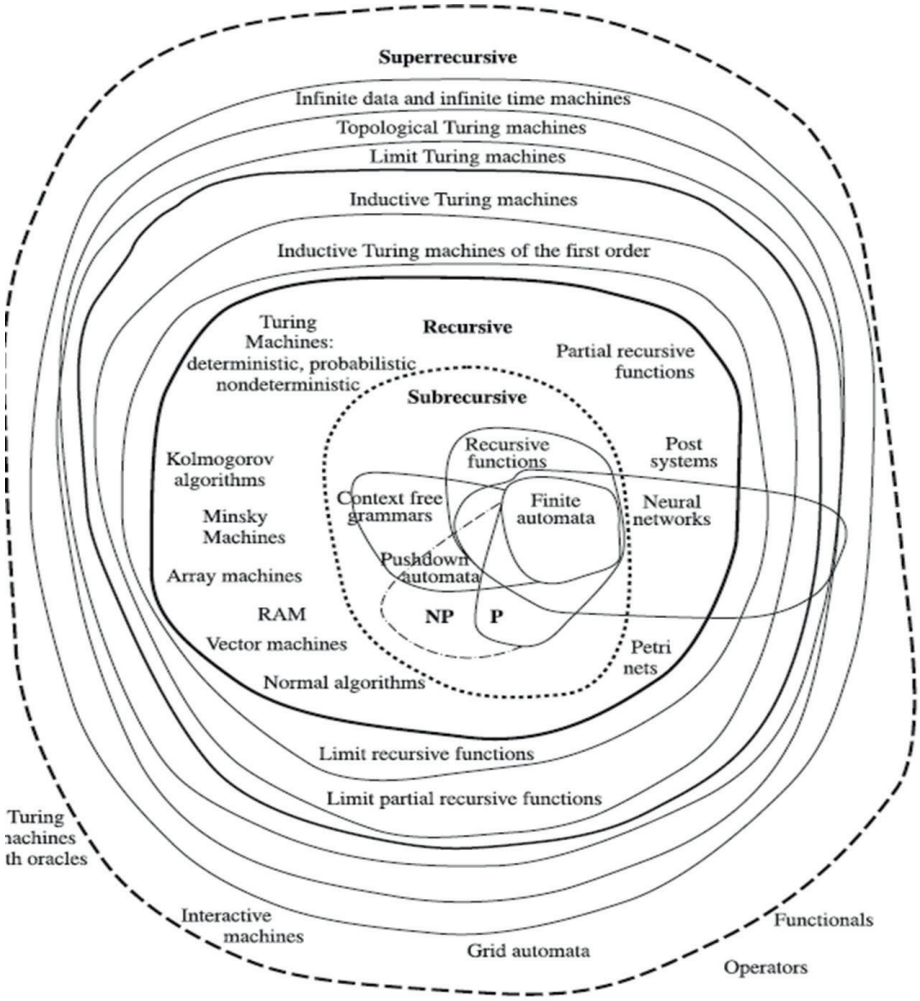

Figure 6.1. Algorithmic universe.

It seems that the cardinal number of this class is countable, although there is no proof here.[15] The task is to distinguish, within the set of all functions defined

[15] Here we supply a proof of a similar proposition (for which we are indebted to David McCarty): If one includes Turing machines alongside oracles (as in the diagram above), then there are an uncountable number of functions thus computed, since there are as many oracles as there are sets of natural numbers. The question of the number of this class of algorithm is indeed a peculiar one. Another peculiar counterexample to this countability would be that one syntactic description can correspond (uncountably?) to an infinite number of different interpretations, but this is a very specific problem and, in our opinion, inconvenient one, as it is in opposition to the usual approach to these matters. It may also be a category in the sense of category theory.

over the natural numbers, the class of computable functions that are of interest to computer science. A similar problem arises with the class of all algorithms.

4. Church's thesis as the foundation of computer science

We will use Shapiro's idea, which refers here to two fundamental principles of computer science: the Church-Turing Thesis (CT) and the thesis of analyzability (TA), which states, "[E]ach procedure can be analyzed by means of basic actions and a set of control structures."[16]

Shapiro's formulation of CT requires a sort of clarification. Church's Thesis itself was formulated by Alonzo Church in the abstract to his article "An Unsolvable Problem of Elementary Number Theory," which was published in 1936. The formulation of the thesis by Church himself is ambiguous and should be explained; cf. (Olszewski 2009, ch. 5). The strongest version of the CT expresses the 'identity' of two concepts taken by Church as definiendum and definiens (CT1).[17] The following theses are weaker: concerning the equivalence of the two concepts (CT2) and the identity of the extensions of the concepts (CT3). The relationship between them is such that CT1 entails CT2 and CT2 entails CT3. CT has a structure that can be generally written as:

[Schema of CT] (I = S)

where 'I' is an intuitive concept, 'S' is an exact concept, and the sign of identity is to be understood as ambiguous, depending on the interpretation of CT as either:

a) the identity of the concepts;

b) the equivalence of the concepts;

c) the identity of the extensions of the concepts.

Following Kreisel, we distinguish between three *versions* of CT by replacing 'I' in the general schema of CT: the intuitive concept of an effectively calculable function (E), the concept of a calculable function (F) by a physical process, and the concept of a mechanically calculable function (M). Therefore, we have three versions:

I. human version (E = R);

II. physical version (F = R);

III. mechanical version (M = R).

However, further variants of CT are those formulations in which the right side of the identity is replaced by an alternative concept from one of these versions. For example, when we replace R with the concept of a function calculable by a Turing machine (T), we obtain the human version variant (E = T), which constitutes the Turing Thesis. Shapiro writes that the Church-Turing Thesis "implies that

[16] These two principles are formulated in the work of Shapiro entitled "Some fundamental principles of computer science". This means that, in his opinion, there may be other such fundamental principles.

[17] Church understood CT, at least at some point in time, as a definition. The issue of the 'identity' of concepts is a complicated one and has been considered in Olszewski's work, sections 5.1.4 and 5.1.5. It is most likely a definition of the analytical type and therefore has the property of logical value.

a computer can carry out any procedure that any mechanistic device, manufactured or naturally grown, could carry out" (Shapiro 2014, 2).[18] This is true, but it pertains to the physical and mechanical version of CT, as well as the additional (logical) premises. The empirical nature of CT should be clearly discerned from the above. On their left side, all three versions contain concepts referring to the actual state of the universe inreside, and, more precisely, to the (computational) capabilities of the human mind, the (computational) capabilities of physical processes, and mechanical procedures (understood in the general sense, also allowing for theoretical machines). This matter is quite complicated and would require more consideration. Some aspects have already been presented in an earlier work by (Olszewski 2009). At the present moment, all versions of CT are considered to distinguish a well-defined set of functions identical to the class of (partially) recursive functions R in the Func class of all functions defined over the natural numbers ($(R \subset Func) \wedge \neg(R = Func)$). This is a crucial determination, since the findings in regard to class Alg are derived from it, as the following condition shows.

A. $a \in$ Alg if and only if$\exists f \in$R (algorithm *a* calculates *f*)

which distinguishes the well-defined set of algorithms Alg from the broader class of algorithms.

The next step is to distinguish a class of computer programs, i.e., a set Prog, on the basis of the class Alg. Then, the classes Real and Machine are to be distinguished on the basis of Prog, in the following manner.

B. $p \in$ Prog if and only if $\exists a \in$ Alg (*p* realises *a*);

C. $r \in$ Real if and only if $\exists p \in$ Prog (*r* is a process of computation generated by the execution of *p*);

D. m $\in$ Machine if and only if $\exists r \in$Real (*m* is the appropriate operational environment for *r*) (Eden 2007, 139).[19]

As can be seen from the above considerations, the key issue in any version is the CT case, which distinguishes a fundamental class of functions that are effectively calculable. Kleene has already drawn attention to the phenomenon of the extraordinary stability of this basic class. Its stability is based on the fact that no matter how we try to characterize a class of effectively calculable functions in the intuitive sense, we always arrive at the class R. Due to this class, we can determine the appropriate algorithm class (Alg), then the program class (Prog), and, consequently, the class of machines (computers) (Machine) and realizations (Real). Correlations between the (R), (Alg) and (Prog) classes are not one-to-one, because e.g., one function can be calculated by many algorithms, and an algorithm can be executed by multiple programs.

[18] Shapiro rightly makes mention of a "mechanistic device (…) manufactured or naturally grown", because a computational machine can really be an artifact or something natural. This is precisely the complex thread I mentioned in the footnote when writing about machines as mainly artifacts.

[19] Eden writes: "The term program-process is thus reserved to that entity which is generated from executing a program-script in the appropriate operational environment". These can also be other real objects, such as the procedures we have assigned to Shapiro.

This situation can be expressed equivalently in the form of a single sentence: $m \in$ Machine if and only if $\exists r \exists p \exists a \exists f ((f \in \mathrm{R}) \wedge (a \in \mathrm{Alg}) \wedge (p \in \mathrm{Prog}) \wedge (r \in$ Real) $\wedge$(algorithm a calculates f) $\wedge$ (p realises a) $\wedge$ (r is a process of computation generated by the execution of p) $\wedge$ (m is the appropriate operational environment for r)). Comparing, for example, class R with class Alg, one can write: $\forall f \exists a$ ($f \in \mathrm{R} \rightarrow (a \in \mathrm{Alg} \wedge a$ calculates $f)$), similarly for sentences expressing relations between the remaining sets of objects from the above list. The very simple fact that the machine belongs to the Machine class, as it seems, has strong ontological commitments, expressing this in Quinean terminology. The same is to be expressed in a simple phrase: CT defines the boundaries of computer science. For this to be the case, it would be necessary to add to the additional equivalences listed above: $f \in$ R if and only if $\exists f \in$ E (f is effectively calculable in an intuitive sense by the human mind (Subject)).[20]

5. A brief digression on the matter of effectiveness

What is of crucial importance here is the proper understanding of the *effectiveness* of a function, i.e. its *executability*, *producing a result that is wanted* etc.[21] It is not the property of the function itself, but the property of the computability of a function that is assigned to functions by the mind (*intellectus*) of the computing person. The phrase "effectively calculable function" can be abbreviated as EC(f) using the following abbreviations: E - effective; C - calculable; and f - function in natural numbers. It would seem that the following equivalences are at play: EC(f) (E $\wedge$ C)(f) $\equiv$ (E(f) $\wedge$ C(f)). If this were the case, we would have an EC(f) $\rightarrow$ E(f), which in general does not have to be the case, particularly when we consider the fact that, according to the Merriam-Webster Dictionary, effective means *created* (*ready*), *specified*, *final* and *intended* (effect) and, when applied to the recursive function of Ackermann, it reveals that this implication is false. In reality, the Ackermann function effectively calculable, which means that while one may theoretically calculate all of the values of its arguments separately, in practice this

[20] The manner in which these considerations are presented utilizes formalism. Are they strictly necessary here? We have chosen to include them for two main reasons: firstly, these formalisms are elementary in nature and help to understand the point being made; secondly, their use is not entirely unfounded, since the variables related to quantifiers, for example, run according to well-defined classes of objects.

[21] The Merriam-Webster Dictionary (on-line) supplies the following definitions for *effect* (here only terms important for our deliberations are included): 1. a change that results when something is done or happens; 2. an event, condition, or state of affairs that is produced by a cause; 3. a particular feeling or mood created by something; 4. an image or a sound that is created in television, radio, or movies to imitate something real. The same dictionary gives for *effective*: 1. producing a result that is wanted; 2. having an intended effect of a law, rule, etc.; 3. in use; 4. starting at a particular time. The fuller definition for *effective* (the first known use comes from the 14th century): 1. producing a decided, decisive, or desired effect <an *effective* policy>; 3. ready for service or action <*effective* manpower>; 4. actual <the need to increase *effective* demand for goods>; 5. being in effect: operative <the tax becomes *effective* next year>.

is ineffective since relatively small arguments are accorded very large values. For clarity's sake, this should be briefly restated as it is genuinely shocking for some: there are functions which can be effectively calculated but they are ineffective if we adopt the dictionary definition of the term.

6. Exceptions according to Shapiro

In Shapiro, we can find an overview of operating systems and heuristic procedures (Shapiro 2014, 1). He believes that they do not fall under the dictionary definition of 'algorithm,' because the former do not converge, and the latter "do not guarantee the correct answer," even though they are of interest to computer science. It seems that this viewpoint may be viewed with some degree of doubt, since algorithms that do not converge are well-defined algorithms, and they are supposed either to fail to converge or stop as a result of a special command dependent on the will of a person. This issue connected with operational programs is encapsulated succinctly in the following work (Enderton 2010, 6): "One person's program is another person's data." The issue of heuristic procedures is somewhat more complex. Harel uses the term 'heuristics' and defines it as 'practical rules.' which permits us to remove from the work of the algorithm the task of searching for cases that are unreliable from the point of view of achieving the goal for which the algorithm was designed. To this end, he mentions programs playing chess as an example of the use of heuristics; cf. (Harel 2000, 366). Let us consider a situation where we have program P with a very high computational complexity, which realizes a certain objective (goal). Such a program is usually practically non-conputable, i.e., it cannot be used, sometimes even for low input values, because the time needed to wait for the result (output) is very long. It is often connected with the number of possibilities that the program has to check, and their number has as an exponent, e.g., the number 1000 (cf. Harel 2000, 366). However, when an intelligent person (specialist) analyses the operations of a program, they may realize that the program sometimes searches through cases where, for example, a solution will definitely not be found or will most likely not be found. Such an observation of the operation of the program is of a practical nature, but more importantly, it is of an *intentional* nature, because it refers to the content (sense) of what the program does, and not only to the purely syntactic manipulation of symbols that constitutes the basis of the way in which algorithms are defined.[22] After the formulation of several such important heuristics in reference to P, it is possible to modify the program itself by providing it with the detected heuristics, obviously using the requisite programming language. The result is a program P', a version of the earlier P program. It is a well-written program, i.e., an element of the Prog class, and it calculates a certain algorithm. However, from the point of view of the purpose for which P has been written, P' has some flaws in that it does not guarantee an

[22] This is a fascinating issue that requires a separate and comprehensive study.

optimal solution and sometimes it will not provide a solution at all, even though such a solution exists for P. Looking at heuristics and heuristic procedures from the perspective of the theory of the mathematical subject (Olszewski 2009, ch. 2) where the Platonic Subject,[23] the Transcendental Subject, and the Empirical Subject are different, it can be said that heuristic procedures are an attempt by the Transcendental Subject to provide the Empirical Subject with the possibility of making a quasi-calculation of what is essentially incalculable. One may also venture the assertion that for any given program P there exists a heuristic program (let us call it the intentional program P") such that, thanks to P", we will obtain essentially the same as program P.[24]

7. The second fundamental principle of computer science

The second fundamental principle of computer science, according to Shapiro, is that "[w]e can analyze any procedure into its basic actions and a set of control structures, which specify how the basic actions are combined." (Shapiro 2014, 2). He refers to the work of Corrado Böhm, and Giuseppe Jacopini (1966), in which a general formulation of computer science was made where a certain calculation model is indicated in which three control structures are listed in addition to the basic activities, and by means of which every calculation procedure can be produced (Shapiro 2014, 2). The adoption of a given set of basic actions, which Shapiro does not list, is crucial given that it may lead to changing the system's calculation power. The control structures are the following: "1) perform one operation, then the next, then the next, etc. (sequence - one after the other; added by me A.O.), 2) select one operation or another, depending on a certain condition (selection - choice; A.O.), 3) loop: repeat the operation as long as a certain condition is fulfilled (loop - repeat; A.O.)" (Shapiro 2014, 2). Odifreddi discusses Böhm and Jacopini's claims in the context of flowcharts (diagrams of sequences of actions) as well as the problem of whether unstructured or structured flowcharts are more convenient or beneficial. By virtue of Wang's findings, we know that any recursive function is calculable in the context of flowcharts, and that the reverse is also true. However, Böhm and Jacopini evidenced an important claim which was previously uncertain, as a result of which it emerges that both types of flowcharts calculate the same class of functions (cf. Odifreddi 1989, 70). To summarize this fragment, it can be said that flowcharts are equivalent to any theoretical model of computability, which is itself an important result. One may also wonder why Shapiro attaches such importance to this calculation model, in addition to the one indicated on the basis of Odifreddi. Perhaps the following argument could be used to clarify this point:

- If something is an object of computer science, then it has a counterpart in the form of a partially recursive function (from CT).

[23] The calculation is an expression of some ignorance, therefore the ideal Platonic Subject does not calculate. He simply knows the value of the function for the argument.

[24] Perhaps this one of the variants of the famous P = NP problem

• Each value of a partially recursive function can be calculated with a single use of the operation minimum applied to a primitive recursive function (from Kleene's Normal Form Theorem).

• The operation minimum is a certain type of loop; the primitive recursive functions are defined by their composition (sequence) and recursion (kind of selection + iteration) (from the relevant definitions).

• Therefore, every computer object can be obtained somehow with the help of the control structures listed by Shapiro.[25]

8. Conclusions and a word in closing

Finally, it is worth noting that a certain group of computer scientists is trying to extend the Func class of computable functions by presenting algorithms that are intended to compute functions outside of the Func class, or attempt to show machines that can calculate such functions.[26] Until now, these intentions have not resulted in practical applications, although until a proof for Church's Thesis is given, we will never have the certainty that this is impossible. This also complicates the epistemological situation of the fundamental problem of the theory of undecidability - the halting problem - as formulated by Turing.[27] Here, we distinguish between the general problem of halting and the problem of halting in Turing. The general halting problem for Turing machines is the question: "Is there an effectively calculable procedure in the intuitive sense that decides the halting problem for Turing machines?" The halting problem for Turing machines is as follows, "Is there a Turing machine which decides the halting problem for Turing machines?" The answer to the second question would be a precisely proven theorem about Turing machines. However, the answer to the general question of the halting problem is not known without assuming the truth of Church's Thesis. The halting problem plays a fundamental role in the theory of undecidability, since the undecidability of many problems is demonstrated indirectly by reducing the issue of their decidability to that of the halting problem. That is probably why supporters of the expansion of the (Func) class most often begin by trying to demonstrate whether the halting problem can be determined or not. In conclusion, CT seems sufficient to establish the boundaries of computing and computer science for the time being. However, there is a considerable amount of pressure to transcend these boundaries. There are scientific fields, such as number theory, where similar pressure, sometimes based on practical demands, led to the discovery of other objects such as complex numbers, which have proven of benefit to science in general. Certainly, transcending the current boundaries of computer science would be akin to a scientific revolution. *We need only to keep an open mind.*

[25] We owe this argument to J. Mycka.

[26] This applies to a sizable group of people, such as Burgin.

[27] The Polish logician, Józef Pepis, wrote about the uncertainty of the undecidability of first-order logic in his PhD. dissertation in the 1930s. He felt it was uncertain because it depends on the correctness of Church's Thesis, which he held is empirical in nature.

References:

Böhm, Corrado and Giuseppe Jacopini. 1966. "Flow diagrams, Turing machines and languages with only two formation rules." *Communications of the Association for Computing Machinery* 9: 366-371.

Burgin, Mark. 2005. *Super-Recursive Algorithms*. Springer.

Eden, Amnon H. 2007. "Three Paradigms of Computer Science," 1-30, preprint (Amnon H. Eden. 2007. "Three Paradigms of Computer Science." *Minds and Machines* 17: 135-167).

Enderton Herbert. 2010. *Computability Theory*. Elsevier: Academic Press.

Harel, David. 2000. *Rzecz o istocie informatyki*. Translated by: Zbigniew Weiss and Piotr Carlson. Warszawa: Wydawnictwo Naukowo-Techniczne. English original: David Harel. 1987. *Algorithmics. The Spirit of Computing*. Wokingham: Addison Wesley Publishing Company.

Kielkopf, Charles F. 1978. "The intentionality of the predicate '__is recursive.'" *Notre Dame Journal of Formal Logic* 19: 165-173.

Odifreddi, Piergiorgio. 1989. *Classical Recursion Theory*. North-Holland.

Olszewski, Adam. 2009. *Teza Churcha. Kontekst historyczno-filozoficzny* ("Church's Thesis. Historical and Philosophical Context"). Kraków: Universitas.

Parsons, Charles. 2008. *Mathematical Thought and Its Objects*. Cambridge University Press.

Shapiro, Stuart C. 2014. *Computer Science: The Study of Procedures*. Accessed by: 3.07.2017. http://www.cse.buffalo.edu.

Wang, Hao. 1957. "A variant of Turing's theory of calculating machines." *JACM* (*Journal of the Association for Computing Machinery*) 4: 63-92.

Merriam-Webster Dictionary. Accessed by: 3.07.2017. https://www.merriam-webster.com/dictionary/effective.