Adam Olszewski
Uniwersytet Papieski Jana Pawła II w Krakowie
Katedra Filozofii Logiki

JAKIE SĄ GRANICE INFORMATYKI?

Motto: *Wszystko jest procedurą*

Niniejsza praca stawia sobie za zadanie odpowiedź na postawione w tytule pytanie.[1] Zacznę od wskazania na ważność pytania o granice jakiejś nauki. Wiąże się to z ogólniejszym pytaniem, dotyczącym granic (czy też granicy[2]) nauki wziętej jako całość. Odróżnienie nauki od nie-nauki od wieków zajmowało umysły filozofów, a dokładniej, chodziło o rozróżnienie między nauką i pseudo-nauką. To ostatnie doprecyzowanie jest o tyle istotne, że oprócz nauki i pseudo-nauki istnieją inne typy działalności badawczej, podczas gdy w przypadku dystynkcji nauka i nie-nauka, na takie typy działalności badawczej miejsca już nie ma. Pojęcie granicy jest pojęciem intuicyjnym i używane jest w codziennym życiu. Występuje ono również w obrębie matematyki głównie w dwóch wersjach, które kodują ściśle dwie intuicje potoczne związane z pojęciem granicy. Pierwsza intuicja wyrażona została w postaci aproksymacyjnego ujęcia, jako pojęcia *granicy ciągu*, zdefiniowanego w analizie matematycznej. Drugie pojęcie granicy, związane z rozdzielaniem obszarów, obecne jest w topologii. W niniejszej pracy skorzystam z intuicyjnych własności granicy, nawiązujących do topologicznego ujęcia tego pojęcia. Tak rozumiana granica, jest granicą pomiędzy sąsiadującym obszarami, które leżą zawsze w jakiejś niepustej przestrzeni. Dlatego dla naszych celów, kluczowe będzie ustalenie wskazanej przestrzeni, w której można będzie wyznaczyć granice informatyki. Przy takim podejściu, znane są co najmniej dwie taktyki: pierwsza – syntaktyczna – za przestrzeń wyjściową bierze język, zaś druga – semantyczna – za przestrzeń bierze klasę jakichś obiektów. Druga z tych taktyk wydaje się być bardziej naturalna i właściwsza, dlatego pójdziemy tropem. Problemem, który się tutaj pokaże jest to, że jeśli zazwyczaj dla teorii istnieje jedno uniwersum, to dla informatyki sprawa jest nieco bardziej złożona. Nie jest jednak jakaś szczególnie szokująca sprawa, gdyż np. logika rozpatrywała od około sześćdziesięciu lat tak zwane *many-sortet logics*, to znaczy systemy z wieloma uniwersami. Shapiro próbował wyróżnić uniwersum informatyki przez wskazanie

[1] Punktem wyjścia dla moich rozważań jest artykuł Stuarta C. Shapiro „Computer Science: The Study of Procedures". Posługuję się polskim tłumaczeniem i odwołuję się do stron z polskiego wydania. W tłumaczeniu błędnie podano pierwszą literę drugiego imienia Shapiro; jest S., a powinno być C.
[2] Osobnym i ciekawym zagadnieniem jest to, dlaczego używa się zwrotu *granice nauki* zamiast *granica nauki*.

zbioru wszystkich *procedur*, gdzie za słownikiem Webstera[3], procedurą jest „szczególny, konkretny sposób postępowania w celu osiągnięcia czegoś”[4] (w oryginale: „*a particular way of doing or of going about the accomplishment of something.*”). Dla niego procedury „nie są obiektami naturalnymi”, ale są „zjawiskami naturalnymi, które mogą być i rzeczywiście są obiektywnie mierzalne – przede wszystkim w terminach potrzebnego im czasu (dotyczy to tych procedur, które się kończą) i w terminach ilości zasobów, których wymagają”[5]. W konsekwencji, takiego rozumienia procedur, uważa on, że informatyka jest nauką przyrodniczą. Takie rozumienie uniwersum (dziedziny) informatyki nie jest powszechne, wymaga ono jednak analizy, gdyż jest nieprecyzyjne, a nawet nieprawidłowe. Przytoczona przez Shapiro definicja procedury zawiera w sobie termin *sposób postępowania*, który dla procedur ustanawia rodzaj bliższy. Należy zwrócić uwagę na to, że *sposób postępowania* jest w ogólności czymś różnym od *postępowania*. To co można empirycznie zaobserwować i zbadać, to są raczej postępowania, a nie sposoby postępowania i to właśnie te pierwsze są zjawiskami naturalnymi, zaś drugie nimi nie są. Postępowania są raczej realizacjami, czy też egzemplifikacjami sposobu postępowania. Ta dystynkcja nawiązuje do tradycyjnych filozoficznych problemów -- uniwersalia *vs* indywidua, czy też types *vs* tokens. Sposoby postępowania są natomiast raczej obiektami abstrakcyjnymi, niźli zjawiskami naturalnymi, jak chce Shapiro. Przez to eksperymentalny charakter informatyki jest wątpliwy, jeśli chcielibyśmy go wywieść ze sposobów postępowania. Natomiast eksperymentalny charakter informatyki da się wywieść z badań nad postępowaniami, które mogą być badane intersubiektywnie. Opiszmy te sprawę nieco dokładniej, gdyż ontologia informatyki jest dość bogata. Od strony ontologicznej mamy, co najmniej, pięć rodzajów obiektów: funkcje (efektywnie) obliczalne (Funk), algorytmy (sposoby postępowania) (Alg), programy komputerowe (w pewnym aspekcie sposoby postępowania) (Prog), realizacje (postępowania) (Real) oraz maszyny (Machine).[6] Sposób istnienia, czy też status ontologiczny, tych obiektów jest różny. Dwa pierwsze typy obiektów najczęściej są rozumiane jako abstrakcyjne tzn. jako pozaczasowe, pozaprzestrzenne i pozaumysłowe. Natomiast status ontologiczny programów komputerowych jest dość skomplikowany. Wyraża to na przykład tzw. „dwuznaczność Fetzera” wskazująca na dualną naturę programów komputerowych. Z jednej strony program „[…] jako ciąg instrukcji zapisanych w języku programowania”[7] ma charakter abstrakcyjny, a z drugiej strony program zapisany na jakimś konkretnym nośniku jest bytem fizycznym.[8] Niektórzy formułują to w ten sposób, że programy komputerowe dzielą się na *skrypty* i *boty*, gdzie te pierwsze są programami zapisanymi w języku programowania, zaś drugie to „obiekty powstające przez uruchomienie skryptu w konkretnych warunkach fizycznych […] które są w

[3] Por. odpowiednie hasło w [4].
[4] Shapiro, 22.
[5] Shapiro, 22.
[6] Ciekawym jest to, że *informacja* nie jest przedmiotem zainteresowania informatyki. Można jedynie powiedzieć, że informatyka zajmuje się technicznymi sposobami przetwarzania informacji. Przykładowo w znanej książce Harela termin *informacja* nie występuje nawet w skorowidzu. Zapisanymi skrótami oznaczam zbiory odpowiednich obiektów.
[7] [2], 52.
[8] Por. [2], 52—53. W tej pracy znajduje się obszerna dyskusja nad statusem ontologicznym programów komputerowych. Podwójna natura programów komputerowych tzn. ich konkretno-abstrakcyjny wymaga odrębnej ontologii. Parsons dopuszcza istnienie takich obiektów nazywając je *quasi-konkretnymi* i jako przykład podaje *sentence-types*. Por. Parsons par. 7.

odróżnieniu od skryptów bytami czasowymi".[9] Te stwierdzenia pokazują, że przyjęte uprzednio stanowisko filozoficzne jest istotne w rozważaniu ontologii informatyki. Im bogatszą ontologię założymy na początku, tym więcej możemy powiedzieć o bytach, które należą do ontologii informatyki i w niniejszej pracy taką bogatą ontologię platońska się zakłada. Na przykład naturalizm, w wersji skrajnie ontologicznej, będzie miał problemy z opisem i wytłumaczeniem pewnych zjawisk. W przypadku naturalizmu, czy też innej postaci redukcjonizmu byty takie jak funkcje czy też algorytmy trzeba będzie imitować za pomocą bytów, których istnienie dana ontologia dopuszcza. Założone tutaj stanowisko filozoficzne pozwala na wskazane już rozróżnienie dwóch 'warstw' rzeczywistości: rzeczywistość fizyczną oraz rzeczywistość abstrakcyjną (idealną). Te dwie warstwy obiektów istotnie się różnią własnościami obiektów, które zawierają. Funkcje określone w liczbach naturalnych tworzą zbiór, który jak wiemy z rozważań teoriomnogościowych, jest mocy continuum. W nim należy wyróżnić podzbiór funkcji obliczalnych, których jest przeliczalnie nieskończenie wiele, ze względu na finitystyczność ich opisu.[10] Drogą sprawą jest to, że klasa wszystkich algorytmów jest również bardzo obszerna, co obrazuje poniższy rysunek pochodzący z pracy Burgina.[11]

[9] Por. [2], tamże. Pogląd taki uznają Ammon Eden i Raymond Turner. Wydaje się jednak, że skrypty nie całkiem dokładnie odpowiadają rozumieniu programu komputerowego jako bytu abstrakcyjnego w rozumieniu Fetzera.
[10] Ta sprawa została bardzo wnikliwie opisana przez Kielkopfa.
[11] Burgin, 250.

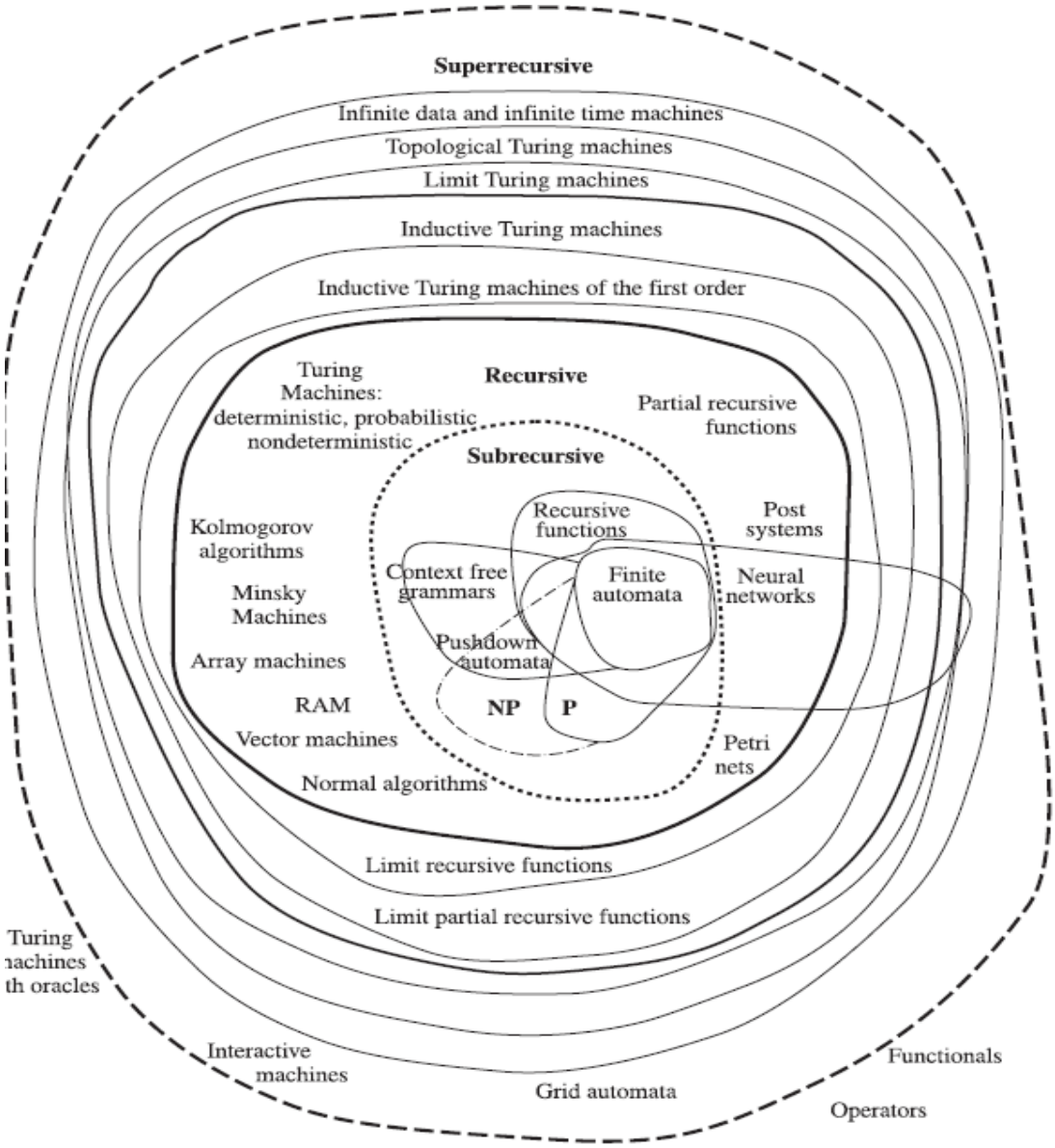

Figure 6.1. Algorithmic universe.

Klasa ta przeliczalna, jak się wydaje, choć tutaj nie ma dowodu.[12] Zadaniem jest wyróżnienie, w zbiorze wszystkich funkcji określonych w liczbach naturalnych, klasy funkcji obliczalnych, będących przedmiotem zainteresowania informatyki. Podobny problem powstaje w odniesieniu do klasy wszystkich algorytmów. Wykorzystamy pomysł Shapiro, który odwołuje się tutaj do dwóch fundamentalnych zasad informatyki: tezy Churcha-Turinga oraz tezy o analizowalności (TA) mówiącej, że „[…] każdą procedurę można analizować poprzez działania bazowe i zbiór struktur kontrolnych".[13] Sformułowanie Shapiro donoszące się do TC wymaga pewnego komentarza porządkującego. Sama teza Churcha (TC) została sformułowana przez Alonzo Churcha w abstrakcie (1935) artykułu „An Unsolvable Problem

[12] Tak kwestia jest dość dziwna. Jednym z przykładów jest to, że jednemu syntaktycznemu opisowi może odpowiadać nieskończenie wiele interpretacji. To jest problem szczególny

[13] Shapiro, 23. Te dwie zasada są sformułowane w pracy Shapiro w paragrafie zatytułowanym „Niektóre fundamentalne zasady informatyki". Znaczy to, że według niego mogą istnieć jeszcze inne takie fundamentalne zasady.

of Elementary Number Theory", który ukazał się w roku późniejszym. Sformułowanie tezy przez samego Churcha jest niejednoznaczne, dlatego należy dokonać pewnych wyjaśnień.[14] Najmocniejsza wersja TC wyraża 'identyczność' dwóch pojęć wziętych przez Churcha jako definiendum i definiens (TC1).[15] Słabszymi są tezy: o równoważności obu pojęć (TC2) oraz teza o identyczności ekstensji pojęć (TC3). Pomiędzy nimi zachodzą takie związki, że z TC1 wynika TC2, a z niej wynika TC3. TC posiada strukturę, którą ogólnie można zapisać (I = S), gdzie 'I' oznacza pojęcie intuicyjne, 'S' pojęcie ścisłe, zaś znak identyczności należy rozumieć jako wieloznaczny, w zależności od interpretacji TC tzn. jako identyczność pojęć, ich równoważność czy też identyczność ekstensji. Idąc za Kreislem odróżniamy trzy *wersje* TC powstające przez zastąpienie I w ogólnej strukturze TC przez: intuicyjne pojęcie funkcji efektywnie obliczalnej (E), pojęcie funkcji obliczalnej przez proces fizyczny (F) oraz przez pojęcie funkcji obliczalnej mechanicznie (M). Mamy zatem trzy wersje: *human version* (E = R), *physical version* (F = R) i *mechanical version* (M = R). Natomiast wariantami TC są te sformułowania w których zamieni się prawą stronę identyczności przez inne pojęcie, w którejś z wersji. Na przykład kiedy zastąpimy R przez pojęcie funkcji obliczalnej za pomocą maszyny Turinga (T), uzyskamy wariant *human version* o postaci (E = T), i to właśnie jest teza Turinga, czyli wariant pierwszej wersji TC. Shapiro pisze, że TC „[…] implikuje, że komputer jest w stanie wykonać każdą procedurę, która może być wykonana przez jakiekolwiek urządzenie mechaniczne, wytworzone czy też naturalnie powstałe"[16]. Tak jest rzeczywiście, ale wynika to z fizycznej i mechanicznej wersji TC oraz dodatkowych przesłanek. Z powyższego przedstawienia powinien być jasny empiryczny charakter TC. Wszystkie trzy wersje mają po swej lewej stronie pojęcia odnoszące się do stanu faktycznego wszechświata w którym żyjemy, a dokładniej do: możliwości (obliczeniowych) umysłu człowieka, możliwości (obliczeniowych) procesów fizycznych i mechanicznych procedur (pojmowanych najogólniej, dopuszczając również maszyny teoretyczne).[17] Na dzień dzisiejszy uważa się, ze wszystkie wersje TC wyróżniają w klasie Funk dobrze określony zbiór funkcji, który jest identyczny z klasą funkcji (częściowo) rekurencyjnych. To ustalenie jest fundamentalne, gdyż ustalenia odnośnie do klasy Alg jest już jego pochodną, ponieważ zachodzi warunek: $a \in$ Alg wtedy i tylko wtedy, gdy $\exists f \in$ Funk (algorytm a oblicza f), który wyróżnia w klasie algorytmów dobrze zdefiniowany zbiór Alg.[18] Następny krok polega na wyróżnieniu w klasie programów komputerowych zbioru Prog w następujący sposób: $p \in$ Prog wtedy i tylko wtedy, gdy $\exists a \in$ Alg (p realizuje a). Jak widać z powyższych rozważań, kluczowa jest sprawa TC w dowolnej wersji, wyróżniająca fundamentalną klasę funkcji efektywnie obliczalnych. Już S.C. Kleene zwracał uwagę na niezwykłe zjawisko stabilności tej podstawowej klasy. Obojętnie w jaki sposób będziemy

[14] Te rozważania oparte są na pracy Olszewski, rozdz. 5.
[15] Church pojmował TC, przynajmniej w pewnym okresie, jako definicję. Kwestia 'identyczności' pojęć jest sprawą skomplikowaną i była rozważana w pracy Olszewski, sekcje 5.1.4 oraz 5.1.5. Kolejny problem to jakiego rodzaju definicją jest TC. Najprawdopodobniej definicją typu analitycznego i dlatego przysługuje jej własność posiadania wartości logicznej.
[16] Shapiro, 23.
[17] Ta sprawa jest dość skomplikowana i wymagałaby dłuższych rozważań. Niektóre zostały przedstawione w pracy Olszewski.
[18] Sens tego sformułowania powinien być jasny. Oczywiście nie jest to definicja lewej strony równoważności, lecz pewne stwierdzenie.

próbowali scharakteryzować klasę funkcji efektywnie obliczalnych, zawsze uzyskujemy klasę Funk. Dzięki tej klasie możemy określić odpowiednią klasę algorytmów, a następnie klasę programów, w konsekwencji można określić klasę maszyn (komputerów). Odpowiedniości pomiędzy klasami Funk, Alg i Prog nie są wzajemnie jednoznaczne, gdyż np. jedna funkcja może być obliczana przez wiele algorytmów, a jakiś algorytm może być zrealizowany przez wiele programów. U Shapiro znajdujemy uwagę dotyczącą systemów operacyjnych oraz procedur heurystycznych.[19] Uważa on, że nie podpadają one pod słownikową definicję algorytmów, gdyż te pierwsze się nie zatrzymują, zaś drugie „nie gwarantują otrzymania poprawnej odpowiedzi", choć są przedmiotem zainteresowania informatyki. Wydaje się, że jest to pogląd chybiony, gdyż algorytmy, które nie kończą pracy są dobrze zdefiniowanym algorytmami, i właśnie mają nie kończyć działania.[20] Kwestia procedur heurystycznych jest nieco bardziej złożona. Harel posługuje się terminem *heurystyk* i określa je jako 'zasady praktyczne', które pozwalają na odrzucenie w pracy algorytmu przeszukiwania przypadków będących nierelewantnymi z punktu widzenia realizacji celu dla którego skonstruowano algorytm, a jako przykład zastosowania heurystyk podaje programy grające w szachy.[21] Weźmy taką sytuację, że mamy jakiś program P o bardzo dużej złożoności obliczeniowej, który realizuje pewien określony cel (zdanie). Program taki jest zazwyczaj praktycznie niekonsumowalny, tzn. nie można z niego skorzystać, czasem już dla małych wartości na wejściu, gdyż np. czas oczekiwania na wynik jest bardzo długi.[22] Kiedy jednak inteligentny człowiek analizuje działanie programu, może się zorientować, że czasami program przeszukuje takie przypadki gdzie np. na pewno nie znajdzie rozwiązania lub raczej na pewno go nie znajdzie. Taka obserwacja działania programu ma charakter praktyczny, ale, co ważniejsze, ma charakter *intensjonalny*, gdyż odwołuje się do treści tego co robi program, a nie tylko do czysto syntaktycznej manipulacji, która jest podstawą definicji algorytmów.[23] Po sformułowania w odniesieniu do P kilku takich istotnych heurystyk, można zmodyfikować sam program wpisując do niego wykryte heurystyki, oczywiście w języku programowania. Uzyskujemy w efekcie wersją P' programu P. Jest ona dobrze napisanym programem, czyli elementem klasy Prog i oblicza pewien algorytm. Jednak z punktu wiedzenia celu dla którego został napisany P, P' posiada wady, gdyż nie gwarantuje optymalnego rozwiązania, a czasem nie daje rozwiązania wcale, choć takie rozwiązania dla P istnieje. Patrząc na heurystyki i procedury heurystyczne z punktu widzenia teorii podmiotu matematycznego[24], gdzie odróżnia się Podmiot Platoński[25], Podmiot Transcendentalny i Podmiot Empiryczny, można powiedzieć, że procedury heurystyczne są próbą obdarzenia Podmiotu Empirycznego obliczalnością tego co jest dla niego istotnie nieobliczalne. Drugą zasadą fundamentalną informatyki wg. Shapiro jest to, że „[k]ażdą procedurę można analizować poprzez działania bazowe i zbiór struktur kontrolnych, który daje specyfikację tego, jak te działania bazowe są

[19] Por. Shapiro, 22.
[20] Takie jest moje zdanie. Być może jest tutaj coś, czego nie rozumiem dobrze.
[21] Por. Harel, 366 i nast.
[22] Wiąże się to często z liczbą możliwości, które musi sprawdzić program, a ich liczba ma w wykładniku np. liczbę 1000. Por. Harel, 366.
[23] Jest to fascynujące zagadnienie wymagające osobnego i obszernego wyświetlenia.
[24] Por. Olszewski, rozdz. V.
[25] Obliczanie jest wyrazem jakiejś niewiedzy, dlatego idealny Podmiot Platoński nie oblicza, po prostu wie jaka jest wartość funkcji dla argumentu.

połączone."[26] Odwołuje się on do pracy Corrado Böhma, Giuseppe Jacopiniego (1966) w której sformułowano ogólne twierdzenia informatyki w którym wymieniono trzy struktury kontrolne, oprócz działań bazowych, z pomocą których można odtworzyć każdą procedurę.[27] Są to następując struktury kontrolne: „1) wykonaj jedną operację, potem następną, potem kolejną itd., 2) wybór: wykonaj jedną operację albo inną, zależnie od pewnego warunku, 3) pętla: powtarzaj daną operację tak długo, jak długo spełniony jest pewien warunek."[28] Odifreddi omawia twierdzenie Corrado Böhma Giuseppe Jacopiniego w kontekście tzw. *flowcharts* czyli diagramów sekwencji działań i problemu, czy wygodniejsze lub korzystniejsze są *unstructered flowcharts* czy też *structered flowcharts*. Na mocy twierdzenia Wanga wiadomo, że każda funkcja rekurencyjna jest obliczalna w sensie flowcharts, prawdziwe jest również twierdzenie odwrotne. Böhm i Jacopini udwodnili ważne twierdzenie, z którego wynika, że oba typy *flowcharts* obliczają tą samą klasę funkcji.[29] Podsumowując ten fragment, można powiedzieć, że *flowcharts* są równoważne dowolnemu teoretycznemu modelowi obliczalności. To ważny rezultat. Na koniec warto zwrócić uwagę na to, że pewna grupa informatyków próbuje rozszerzyć klasę funkcji obliczalnych Funk w ten sposób, że są prezentowane algorytmy, które mają obliczać funkcje spoza klasy Funk, bądź próbuje się wskazać maszyny, które takie funkcje mogą obliczać. Dotychczas te zamierzenia nie przyniosły rezultatu w postaci praktycznych zastosowań, choć do czasu dowodu tezy Churcha, pewności, że jest to niemożliwe nigdy nie będziemy mieli. Jeden z fundamentalnych problemów teorii nierozstrzygalności to *problem stopu*, sformułowany przez Turinga. Odróżnijmy tutaj ogólny problem stopu od problemu stopu Turinga. Ogólny problem stopu dla maszyn Turinga to pytanie: czy istnieje procedura efektywnie obliczalna w sensie intuicyjnym, która rozstrzyga problem stopu dla maszyn Turinga? Zaś problem stopu dla maszyn Turinga ma postać: czy istnieje maszyna Turinga rozstrzygająca problem stopu dla maszyn Turinga? Odpowiedzią na drugie pytanie jest ścisłe twierdzenie o maszynach Turinga. Natomiast odpowiedź na ogólny problem stopu nie jest znana, bez założenia prawdziwości tezy Churcha. Problem stopu pełni fundamentalną rolę w teorii nierozstrzygalności gdyż nierozstrzygalność wielu problemów wykazuje się pośrednio przez zredukowanie kwestii ich rozstrzygalności do rozstrzygalności problemu stopu. Pewnie dlatego zwolennicy rozszerzenia klasy Funk bardzo często próbują wykazać, że problem stopu jest rozstrzygalny. Podsumowując: na dzień dzisiejszy wydaje się, że TC wyznacza granice informatyki w sposób wystarczający. Istnieje jednak dość duża presja na przekroczenie tej granicy. Znane są dyscypliny naukowe, jak na przykład teoria liczb, gdzie podobna presja, mająca czasem oparcie w zapotrzebowaniu praktycznym, doprowadziła do odkrycia innych obiektów jak np. liczby zespolone, i stało się to z pożytkiem dla nauki. Z cała pewnością przekroczenie obecnych granic informatyki będzie swoistą rewolucją naukową. *We only need to keep an opened mind* (R. Gandy).

[26] Shapiro, 23.
[27] Por. Shapiro, 23.
[28] Shapiro, 24.
[29] Por. Odifreddi 70.

Literatura

1. Corrado Böhm, Giuseppe Jacopini, “Flow diagrams, Turing machines and languages with only two formation rules”, *Communications of the Association for Computing Machinery*, 9: 366—371.
2. Izabela Bondecka-Krzykowska, *Z zagadnień ontologicznych informatyki*, Wydawnictwo naukowe UAM, 2016.
3. Mark Burgin, *Super-Recursive Algorithms*, Springer 2005.
4. David Harel, *Rzecz o istocie informatyki*, WNT 2000.
5. Charles F. Kielkopf, „The intentionality of the predicate ‘__is recursive’”, *Notre Dame Journal of Formal Logic*, 19: 165—173.
6. Piergiorgio Odifreddi, *Classical Recursion Theory*, North-Holland 1989.
7. Adam Olszewski, *Teza Churcha. Kontekst historyczno-filozoficzny*, Universitas 2009.
8. Charles Parsons, *Mathematical Thought and Its Objects*, CUP 2008.
9. Stuart C. Shapiro „Computer Science: The Study of Procedures”, http://www.cse.buffalo.edu . Polskie tłumaczenie [w:] *Filozofia informatyki. Antologia*, 21--25, opracowanie i tłumaczenie Roman Murawski, Wydawnictwo naukowe UAM, 2014.
10. *Webster’s Third New International Dictionary*, unabridged (1993).