Some Transfinite Generalisations of Gödel's Incompleteness Theorem

Jacques Patarin

University of Versailles
45 avenue des États-Unis, 78035 Versailles Cedex, France
jacques.patarin@prism.uvsq.fr

Abstract. Gödel's incompleteness theorem can be seen as a limitation result of usual computing theory: it does not exist a (finite) software that takes as input a first order formula on the integers and decides (after a finite number of computations and always with a right answer) whether this formula is true or false. There are also many other limitations of usual computing theory that can be seen as generalisations of Gödel incompleteness theorem: for example the halting problem, Rice theorem, etc. In this paper, we will study what happens when we consider more powerful computing devices: these "transfinite devices" will be able to perform α classical computations and to use α bits of memory, where α is a fixed infinite cardinal. For example, $\alpha = \aleph_0$ (the countable cardinal, i.e. the cardinal of $\mathbb{N}$), or $\alpha = \mathfrak{C}$(the cardinal of $\mathbb{R}$). We will see that for these "transfinite devices" almost all Gödel's limitations results have relatively simple generalisations.

1 Introduction

Gödel's famous incompleteness theorem was first presented on October 7, 1930, at the first international conference of mathematics philosophy, at Königsberg. This result can be seen as a limitation result of usual computing theory: it does not exist a (finite) software that takes as input a first order formula on the integers and decides (after a finite number of computations and always with a right answer) whether this formula is true or false. In 1930 no real computer existed yet, but the mathematical analysis of the functions that can be effectively computed with (finite) software (i.e. "recursive functions") had began. Gödel was studying sets of axioms for which there is an—effective, finite, recursive—computing way to know if a given formula was a member of these axioms or not.

What will happen if we consider more powerful computing devices? For example, if we include in the set of axioms all first order formulas that are true in $\mathbb{N}$ (with the standard interpretation of addition and multiplication) we will obtain a complete set of axioms (i.e. with no undecidable and contradictory formulas); however, it is not possible with a classical software to know if a given formula is one of the axioms or not.

In this paper we will study what happens when we use "transfinite software", i.e. software that can be run on "transfinite computers", generalised computers that can perform α classical computations and use α bits of memory, where α is a fixed infinite cardinal. These transfinite computers are able to compute more than classical computers, but, are they limited by "transfinite questions" that can be seen as generalisations of classical computations questions? In fact, as we will see, it is possible to generalise almost all classical results within this framework. Such generalisation is not totally new. In [6] and in some references mentioned in [6], problems linked to "Totality, Knowledge and Truth", and "Incompleteness" are mentioned, and it is clearly stated that some limitation results can be generalised beyond the classical theory of computation; a relativised version of incompleteness was proved in [3]. It seems, however, that an explicit description of the main limitation theorems presented in the framework of "transfinite computers" has not been done yet.

In [4] it is proved that any Turing machine that uses only a finite computational space for every input cannot solve any undecidable problem even when it runs in accelerated mode (unlike as in this paper where the memory will be infinite). A natural continuation of this work, that we hope to obtain in the near future, is the generalisation of the results of [2] for "transfinite software".

2 A transfinite computing model

2.1 General remarks

We will use the transfinite computing model described in [14]. To make this paper self contained we will explain in this section the model.

It is also important to notice that our limitation proofs and results below are very stable and generally will not depend on the chosen transfinite computing model, as long as the model is reasonably natural and uses sets (working on classes instead of sets may involve a specific analysis which will not be presented in this paper).

2.2 Transfinite computations

Let α be a fixed infinite cardinal. For example $\alpha = \aleph_0$ (the countable cardinal, i.e. the cardinal of $\mathbb{N}$), or $\alpha = \mathfrak{C}$ (the continuum, i.e. cardinal of $\mathbb{R}$). In "α-software" we use "transfinite computers" able to perform α computations with α bits of memory. It is possible to describe precisely this model of computations, see [15], [16], [7], [9], [10], for example. The general idea is to follow a generalisation of the Church's Thesis: as soon as a computation will be clearly feasible with α bits of memory and α computations, we will include it in the model. Moreover, the results of this paper will be very stable with respect to small changes in the infinite computation model.

Readers familiar with Ordinal Turing Machines, (OTM), with tapes whose cells are indexed by ordinals, as described in [9], can just go directly to section 3.

We will speak of "α-programs" or "α-softwares". We can assume that the memory is separated in 4 zones of bits: the input memory, the program memory, the variables of computation memory, and the output memory. Without loss of generality we can assume that the input memory is made of 1, or 2 (or more but $\leq \alpha$) inputs of α bits. The program memory contains a well ordered set of α elementary operations. Thanks to the fact that the program memory is well ordered, we can know at each "time" of the computation which is the next operation to perform. The word "time" is of course here a generalised word: it means that when any set of operations has been performed, we know precisely what is the next operation to be performed. More precisely than "time", it is the succession of some ordinals that we will use. To each operation T at a certain place in the program we will associate an ordinal β, so we can say that T is the operation number β, or of position β. Each elementary operation can be of two kinds: simple, or GOTO. A simple elementary operation is a classic operation present in any computer language (such as C, on two words of 64 bits, for example) such that these two words are chosen at the addresses a and b of the memory and the result is stored at the address c of the memory. Here a and b can be addresses of the input memory, or of the variables of computation memory, c can be an address of the variables of computation memory, or of the output memory; a, b, and c are addresses of at most α bits and are associated with the current operation. So each instruction of the program (operation and its position or number) can memorise a, b, c, and the operation to be performed. Of course, it is possible to use any other classical computer language instead of the C language, or to use words of 32 bits (or another length) instead of 64 bits. This will not change the set of functions that we can compute. A special instruction is the "stop instruction". When this instruction is performed, the program stops and the output of the program is the value stored in the α bits of the output memory. The GOTO operation is an operation of the form (if $X = k$) then GOTO β, where β is an ordinal. Thus this GOTO instruction says that the next instruction to be performed is the instruction number β (or of position β), if a variable X of α bits is equal to the value k of α bits. (Note that β can be any ordinal smaller than the ordinal of the current GOTO instruction performed.) If $X \neq k$, to determine the next instruction we will follow, as for the simple instructions, the usual order of the ordinals of the instructions. It is also possible to describe our model of transfinite computations with generalised Turing machines.

2.3 Coding the instruction ordering

In the α bits of the program memory zone there are various simple ways to describe the ordering (well ordering) of the instructions. Let us give here an example for $\alpha = \aleph_0$. (It is easy to generalise this example for any cardinal α.) Let P be an α program. By definition, we will call "ordinal of P" the ordinal of the (well ordered) set of all instructions of P. For example, if $\alpha = \aleph_0$, this ordinal may be ω, or ω^3. A countable ordinal can be described as a good ordering on $\mathbb{N}$. So each countable ordinal can be written as a set of $\aleph_0$ integers:

for each integer n, we will give the list of all the integers m such that $m < n$ for this ordering. We need for this $\leq \aleph_0 \times \aleph_0$ bits. The "infinite processor" can find the first instruction (no instruction is strictly smaller), and then, at each step, it can check all the integers in order to find the next instruction to be performed.

Remark. A classical result on ordinals says that the set of countable ordinals has cardinal $\aleph_1$ (i.e. the smallest non countable infinite cardinal). We know that $\aleph_1 \leq \mathfrak{C}$. (However we do not know if $\aleph_1 = \mathfrak{C}$ or not, this is the famous undecidable problem called "the continuum hypothesis".) Moreover each real number can be given by $\aleph_0$ bits. Therefore, each countable ordinal can be given by $\aleph_0$ bits. This is what we do here for the ordinal of program P. The method used for $\alpha = \aleph_0$ can also be extended to any infinite cardinal α since, adopting the axiom of choice, for any infinite cardinal α, we have $\alpha^2 = \alpha$.

Example. The function $x \to x^2$ on $\mathbb{Q}$ is a one way function in the model of infinite computation of [17]. In our model of infinite computations, this function however is not a one way function. In order to find a rational (or a real) x such that $x^2 = y$ with $\aleph_0$ computations and $\aleph_0$ bits of memory, we can, for example, find all the bits of x, one by one. If we know that x is a rational number, then we can also try all the rational numbers one by one (card $\mathbb{Q} = \aleph_0$), square them, and check whether we get y. Here again we need "only" $\aleph_0$ computations and $\aleph_0$ bits of memory.

Remark. On classical computers bits can have the value 0, or the value 1. In our model of computation, it is possible to assume that the values can be 0, 1, or "not fixed". The value "not fixed" will be obtained for example when the bit has flipped from 0 to 1 and from 1 to 0 infinitely many times, without being fixed to 0 or 1. However, it is possible to prove that if this value "not fixed" is changed to 0 (or 1), the infinite model of computation will be the same (i.e. we will be able to compute exactly the same functions); in this case the model may be slightly less natural. (A variable B can be at $11\ldots1\ldots$ with an infinity of 1 if and only if a bit b has changed an infinity of times its value.)

3 α-Recursive sets, α-Recursively enumerable sets

We start with a few definitions.

Definition 1 We say that an α-software **stops** or **gives the output after α computations** when the α-software stops after performing at most α computations.

Definition 2 We denote by $I_\alpha = \{0,1\}^\alpha$ the set of all sequences of α bits. Therefore $I_{\aleph_0}$ can be identified with the set $\mathbb{R}$ of all the real numbers, or with $[0,1]$ for example.

Definition 3 Let A be a subset of I_α. We say that A is α-**recursive** if there exists at least one α-software P such that when we give $n \in I_\alpha$ as input to P, then P will be able to answer after at most $\leq \alpha$ operations if $n \in A$ or $n \notin A$.

Definition 4 We say that A is α-**recursively enumerable** if there exists at least one α-software P such that when we give $n \in I_\alpha$ as input of P:

1. if $n \in A$, then P will be able to answer $n \in A$ after at most α operations.
2. if $n \notin A$, then P does not answer after α operations, or P will answer $n \notin A$.

Definition 5 Let f be an application $I_\alpha \to I_\alpha$. We say that f is α-**recursive** if there exists at least one α-software P such that for all $n \in I_\alpha$, when n is given as input to P, P will give the output $f(n)$ after performing at most α computations.

Remark. There are α^α applications from I_α to I_α, and the number of α-softwares is $\leq \alpha$. Since $\alpha^\alpha \geq 2^\alpha > \alpha$ (Cantor Theorem), it follows that some applications are neither α-recursive nor α-recursively enumerable.

Definition 6 Let α be a cardinal. Put $I_{limit\ \alpha} = \cup_{\beta<\alpha} I_\beta$. We define an α **-limit-software** as a α-software such that:

1. The inputs are the elements of $I_{limit\ \alpha}$.
2. The variables of computation memory are in $I_{limit\ \alpha}$.
3. The program memory is in I_β with $\beta < \alpha$.
4. The input is an element of $I_{limit\ \alpha}$.

Definition 7 Let $A \subset I_{limit\ \alpha}$. We say that A is α-**limit-recursive** if there exists at least one α-limit-software P such that when we give $n \in I_{limit\ \alpha}$ as input of P, P will be able to answer after at most β operations, $|\beta| < |\alpha|$, if $n \in A$ or $n \notin A$.

Definition 8 Let $A \subset I_{limit\ \alpha}$. We will say that A is α-**limit recursively-enumerable** if and only of it exists at least one α-limit-software P such that: when we give $n \in I_{limit\ \alpha}$ as input of P,
• if $n \in A$ then P will be able to answer after at most β operations, $|\beta| < |\alpha|$, that $n \in A$.
• if $n \notin A$ then P will not answer or P will $n \notin A$.

Remark.

1. The above definitions are generalisations of the classical definition of recursiveness, i.e. $\aleph_0$-limit-recursive = recursive (usual meaning) and $\aleph_0$-limit-recursively-enumerable = recursively enumerable (usual meaning).
2. The definitions of α-limit-recursiveness may be interesting when α is a cardinal with no predecessor (as $\aleph_0$), i.e. when there is no cardinal $\beta < \alpha$ such that α is the smallest cardinal $> \beta$, because if α has a predecessor β, then α-limit recursiveness is simply β recursively.

4 A generalisation of "recursively enumerable and not recursive sets"

To achieve the main aim of this paper, i.e. to show that that most of the classical limitation results of logic can be generalised in the model of transfinite computations, with almost the same proofs as in the case of classical case, we will follow here one of the classical ways to obtain such limitation results (as in [11] or [13]). Of course, one can possibly obtain the same results following other proofs.

4.1 α-Code of a α-software

To each α-software T we can associate in an injective and "simple" way an element of I_α, called its α-code, and denoted $\lceil T \rceil$. By "simple" we mean that there exists an α-software that takes $\lceil T \rceil$ as input, and then produces the sequence of α instructions of T.

Example: If $\alpha = \aleph_0$, the indices of the instructions of T are countable ordinals, and the set of these indices is countable. They form a well-ordering and this can be seen as a well-ordering on $\mathbb{N}$. Such a well-ordering on $\mathbb{N}$ can be described as follows: to each natural integer we associate the of integers that are smaller than the given integer (for the well-ordering). There are $\aleph_0$ such integers. Then for each elementary instruction, we can associate 2 or 3 real numbers (for example the instruction "if ($X = K$) then GOTO β"). Thus we can associate to such a software T, an injective and "simple" application from $\mathbb{N}$ to $\mathcal{P}(\mathbb{N}) \times \mathbb{R}^3$. Let B be the set of applications from $\mathbb{N}$ to $\mathcal{P}(\mathbb{N}) \times \mathbb{R}^3$. $|B| = |\mathbb{R}|$; there exist "simple" bijections from B to $[0, 1]$. So to each $\aleph_0$-software T, we can associate in an injective and "simple" way an element of $I_{\aleph_0}$ which is its code.

Remark. The above method works for any $\alpha \geq \aleph_0$.

4.2 Software simulation

If B is an α-software and x an element of I_α, we denote by $B(x)$ the result of software B when x is the input: i.e, the value of the output memory (it is also an element of I_α) when the software stops after $\leq \alpha$ operations.

There exists an α-software P which, when it is given $x \in I_\alpha$ as input:

1. P "finds" the α-software X such that $\lceil X \rceil = x$ in case such an α-software X exists. This comes from the fact that the coding is "simple" (cf. above).
2. P executes the same instructions on x as X would execute with x as input. Thus, for all $x \in I_\alpha$, $P(x) = X(x)$ provided there exists an α-software X such that $\lceil X \rceil = x$.

4.3 The basic theorem

Theorem 1 *There exists $A \subset I_\alpha$, such that A is α-recursively enumerable, but A is not α-recursive.*

Proof: Let P be the α-software previously defined such that $P(x) = X(x)$, whenever there exists an α-software X with code x. Let

$$A = \{x \in I_\alpha \mid P(x) \,\text{is computed in}\, \leq \alpha \text{ computations.}\}$$

1. Since A is defined by the α-software P, A is α-recursively enumerable.
2. Assume that A is α-recursive and q is the code of an α-software Q such that:

   $$x \notin A \Leftrightarrow Q(x) \,\text{is computed in} \leq \alpha \,\text{computations.}$$

   Then

   $$q \in A \Leftrightarrow P(q) \,\text{is computed in} \leq \alpha \,\text{computations} \quad \text{(by definition of } A\text{)}$$

   i.e.

   $$q \in A \Leftrightarrow Q(q) \,\text{is computed in} \leq \alpha \text{ computations} \quad \text{(by definition of } P\text{)}$$

   i.e.

   $$q \in A \Leftrightarrow q \notin A \quad \text{(by definition of } Q\text{)}.$$

   This is not possible. Thus A is not α-recursive. □

5 The decision problem, the halting problem and ordinal length of computations for α-softwares

We generalise the undecidability of the above problems for α-softwares.

5.1 The decision problem

Theorem 2 *There is no general algorithm, programmable with α-software, which could, using always $\leq \alpha$ computations, to decide whether a mathematical proposition on elements of I_α is true or not.*

Proof: It is enough to consider all the propositions of the form $n \in A$, where $n \in I_\alpha$, and A is the set defined in Theorem 1 above. Since A is not α recursive, there exists no α-software which, when applied to one of these propositions $n \in A$ can decide, using $\leq \alpha$ computations, whether this proposition is true or false. $\square$

Remark.

1. These mathematical propositions can be written with quantifiers $\forall$, $\exists$, the usual logic symbol and the operators $+$, $-$, $\times$, $\div$ and with $\leq \alpha$ elementary finite formulas. We then get a generalisation of Gödel's Incompleteness Theorem. (We just have to write the α-software with such formulas, which are generalisations of first order classical formulas with α characters, which is always possible).
2. Some properties that are true on sequences of α bits are lost if we are limited to α computations and α bits of memory, for any infinite cardinal α.

5.2 The halting problem

Theorem 3 *There exists no α-software which can decide with $\leq \alpha$ computations whether an arbitrary α-software will stop or not in $\leq \alpha$ operations.*

Proof: If such α-software existed, then we could use it to write an α-software which, when presented with an $x \in I_\alpha$ as input could decide in $\leq \alpha$ operations whether $P(x)$ stops after $\leq \alpha$ computed or not. But A is not α recursive, thus such an α-software does not exist. $\square$

5.3 Ordinal length of computations for α-softwares

Theorem 4 *There exists no general α-software taking as input the code of a program T which stops in $\leq \alpha$ computations, and gives as output an ordinal ω_α such that the cardinal of ω_α is $\leq \alpha$ and the ordinal of the number of computations performed before T stops is $\leq \omega_\alpha$.*

Proof: If such a program exists, we could know with $\leq \alpha$ computations if $n \in A$ or $n \notin A$. This is in contradiction with the fact that A is not α-recursive (A comes from Theorem 1). In that case, it would be sufficient to stop after ω_α computations to conclude that the program does not answer in $\leq \alpha$ computations. $\square$

6 The fixed point theorem on α-softwares

Let $z, x, y \in I_\alpha$, such that there exists an α-software Z with code z and two entries: x and y. We denote $z[x, y]$ the output of the software Z on entries x and y when this software stops in $\leq \alpha$ computations.

Remark. If Z does not stop after $\leq \alpha$ computations, we can consider that $z[x, y]$ is the information "z does not stop" after $\leq \alpha$ computations.

Theorem 5 (Iteration Theorem) *There is an α-recursive application of two variables $s(x, y)$ such that:*

$$\forall z, x, y \in I_\alpha,\ z[x, y] = s(z, y)[x].$$

Proof: We consider the α-software s that performs the following operations when it is given z and y as inputs:

1. Finds the sequence of instructions of the α-software Z with code z. (This is possible since the coding is "simple").
2. Computes the code of an α-software which on input $x \in I_\alpha$ simulates Z on the inputs x and y. (Again this is possible since the coding is "simple").

Then this α-software computes $s(z, y)$ such that

$$\forall z, x, y \in I_\alpha,\ z[x, y] = s(z, y)[x].$$

□

Theorem 6 (Fixed point theorem on α-softwares) *For every α-recursive application h, there exists $e \in I_\alpha$ such that:*

$$\forall x \in \mathbb{N},\ e[x] = h(e)[x].$$

Remark. Every α-software can be written using a program with a single input ($\alpha^2 = \alpha$ since α is an infinite cardinal). Thus the fixed point theorem on α-softwares can be written in the form: If "h is an α-recursive application, there exists always an α-software with code e and an α-software with code $h(e)$ which on any input $x \in I_\alpha$ gives the same output (and does not give any output).

Indeed, let $f(x, y) = h(y)[x]$. Since $s(z, y)$ is α-recursive, there exists d the code of an α-software with computes $f(x, s(y, y))$. For all $x \in I_\alpha$, et $\forall y \in I_\alpha$, we have:

$$f(x, s(y, y)) = \begin{cases} d[x, y], & \text{by definition of } d, \\ d(d, y)[x], & \text{by definition of } s\,. \end{cases}$$

Let $e = d(d, d)$. With $y = d$, we get that for all $n \in I_\alpha,\ f(x, e) = s(d, d)[x] = e[x]$. Thus (by definition of f), $\forall x \in I_\alpha,\ h(e)[x] = e[x]$. □

7 Rice theorem on α-softwares

A function f from D_f to I_α, where $D_f \subset I_\alpha$, is called **α-recursive semi-function** if there exists an α-software which computes $f(x)$ when given an input $x \in D_f$ in $\leq \alpha$ computations, and does not answer in $\leq \alpha$ computations when it is given $x \notin D_f$.

Theorem 7 (Rice theorem on α-softwares) *Let F be a non-empty proper subset of all α-recur- sive semi-functions. Then the set*

$$A = \{n \in I_\alpha \mid n \text{ \emph{is the code of an} } \alpha - \text{\emph{recursive semi-function}} \text{ of } F\}$$

is not α-recursive.

Proof: By definition of F, $A \neq \emptyset$ and $A \neq I_\alpha$. Let a and $\bar{a}$ be two elements of I_α such that $a \in A$ and $\bar{a} \notin A$. We set

$$h : \begin{cases} \quad I_\alpha & \to I_\alpha \\ x \in A & \mapsto \ \bar{a} \\ x \notin A & \mapsto \ a \end{cases}$$

Suppose that A α is α-recursive. Then, h is a α-recursive application. It follows, from the fixed point theorem, that there exists $e \in \mathbb{N}$ such that the α-programs coded by e and $h(e)$ compute the same semi-function. So $e \in A \Leftrightarrow h(e) \in A$ (by definition of A and e). But by definition of h, $e \in A \Leftrightarrow h(e) \notin A$. Thus $h(e) \in A \Leftrightarrow h(e) \notin A$, a contradiction. This shows that A is not α-recursive. $\square$

This generalised Rice theorem shows that there exists no α-software which decides:

1. If two α-softwares compute the same function. (Choose a singleton for F.)
2. If an α-software will always answer 0 on any input. (Choose F that contains only the null function.)
3. If an α-software will always give an answer. (Choose for F the set of recursive semi-functions defined on I_α.)
4. If an α-software will always return values in a given subset B. (Choose for F the set of semi-functions whose output is in B.)

This generalised Rice Theorem shows that the problem of "debugging" an α-software, or the understanding of what a α-software is doing, generally uses more than α computations.

8 Some "philosophical" comments

Some physicists have suggested at the end of the 19th century, and at the beginning of the 20th century the "end of physic": everything that should be discovered in physic was already discovered, or almost already discovered. For example all the forces, elementary particles etc. Of course, these physicists were a minority, and history has so far shown that they were wrong since new fundamental and very important discovery in physics have been done after their claim.

In mathematics, it is difficult to find arguments to claim the end of the work. In fact, a new theorem generally gives new open problems. As we have seen in this paper, this will be also the case if one day we have access to much more powerful computing devices. If one day we are able to perform α computations, where α is a given infinite cardinal, then we will be able to solve a lot of mathematical open questions, but new open questions will naturally appear that cannot be solved with these α devices. In fact, for all infinite cardinal α, the exact possibilities of computing of these α-devices (such as the halting problem for these devices) is out of reach of these devices. Therefore, for all infinite cardinal α these problems appear at the same time to be the natural problems on the power of computation that we have, and undecidable with this power of computation.

Aristotle defined happiness at the "impression to use at best our capacities". This is compatible with large computing capacities, if we do not feel depress from the natural unsolvable questions that will appear...

Theology and computations

It is possible to think (if we do not take it too seriously) about some possible theological implications of such transfinite limitations. In fact, similar comments that mix some mathematical results on the infinite cardinals and "god" have been done before. For example in [5] the religious history in Russia of some interpretation of Cantor theory is presented (it was a bloody history in Stalin soviet state). Gödel itself has published a very weird "proof" of God existence from some strange axioms.

There are in fact two classical "definitions" of God. In the first definition, God is the entity that has created the earth and all the living species in this universe. In the second definition (13th century), that is called the "ontological" definition of God, God is defined as "the most powerful entity that can exist with no contradictions".

With the first definition, it is possible that God exist and is limited by α computations and β bits of memory, where α and β are fixed finite or infinite cardinals (for example $\alpha = \beta = \aleph_0$). Then some purely mathematical questions will not be solvable by computation by such a God despite the fact that these questions can be easily asked with its capacities. (There is maybe however some possibility to access mathematical truth without computing...).

With the second definition the results of this paper leads naturally to the question : "is this ontological definition of God self contradictory ?". An attractive option to save this definition is to use "classes" instead of "sets" as the basic tools. For example we can mathematically define the field of the "surreal numbers" (cf [8]) that contains all the $\mathbb{R}$ fields. This surreal field is a class, not a set unlike all the $\mathbb{R}$ fields. Similarly, we can imagine a God able to performs α computations for all cardinal α. However if we can define software with a class of instructions (and not only a set of instruction), then, again, some new limitation results seems to appear (the halting problem for these class-softwares for example). In [6] Patrick Grim goes as far as to suggest that Truth and Totality may be not compatible. This is perhaps going too far. In fact it seems that we have to choose between Totality (or Universality) and the capacity to create new mathematical objects from previous ones, and we cannot have at the same time these two properties. (A similar problem appears with Russel Paradox, or Cantor theorem: we cannot consider a "universal set" set E since $\mathcal{P}(E)$ would have a cardinal strictly greater than E). Moreover, as we said above, it is maybe possible to have access to truth without computing.

Anyway, if after my death I have access to $\aleph_0$ of happiness I will probably not complain that this is the smallest infinite cardinal.

Remark. At present almost nobody takes these mathematical results of limitation, or paradox of totality, as serious arguments against the possible existence of an all mighty god. However this may change in the future.

In fact, I also do not take these arguments very seriously, but it is however not so easy to avoid them. I will present below some possible ideas that may be used if we want to claim that monotheist religions are not in contradiction with these mathematical limitation results.

1. Maybe the real God, if he exists, do not satisfy the ontological definition. For example, maybe he has created this universe but is limited by a finite number of computations, or by a given infinite cardinal number of computations.
2. Maybe the human proofs are just illusions since God make us believe that these proofs are valid, but they are not. (The fact that some computers can check these proofs does not change anything since God can change the output of these soft wares). It is however difficult to see what would be God motivation for that.
3. Maybe God never ask himself some questions about its own limitations. He has created some species that are less powerful than him and he is able to solve the halting problems of all the computing devices that these species can build.
4. Maybe God can access truth without computing.
5. Maybe nothing really new and "interesting" appears beyond a certain number of transfinite computations. We know that new mathematical results appear each time we increase the transfinite cardinal of possible computations, but maybe these mathematical results are not considered interesting, unlike feelings like love, good or bad actions, responsibility, etc.

9 Conclusion

We have shown that most of the logic limitation results proved in the classical theory of computation can be generalised to the case when the computing devices are able to perform α computations and use α bits of memory, where α is a given fixed cardinal. It is expected that some recent results of limitations, such as those of [2], can also be generalised in this framework. This can be the subject of further work.

References

1. Cristian Calude, Helmut Jürgensen, and Marius Zimand. Is independence an exception? *Appl. Math. Comput.*, 66:63–76, 1994.

2. Cristian S. Calude and Helmut Jürgensen. Is complexity a source of incompletness? *Advances in Applied Mathematics*, 35:1–15, 2005.
3. Cristian S. Calude and Sergiu Rudeanu. Proving as a computable procedure. *Fundamenta Informaticae*, 64(1–4):43–52, 2005.
4. Cristian S. Calude and Ludwig Staiger. A Note on Accelerated Turing Machines. *Math. Struct. in Comp. Sciences*, 20:1011–1017, 2010.
5. L. Graham and J.M Kantor. *Naming Infinity.* The Belknap Press of Harward University Press, 2009.
6. Patrick Grim. *The Incomplete Universe, Totality, Knowledge and Truth.* A Bradford Book, The MIT Press, 1991.
7. J.D. Hamkins and A. Lewis. Infinite time Turing machines. *Journal of Symbolic Logic*, 65(2):567–604, 2000.
8. Donald Knuth. *Surreal Numbers: How two ex-student turned to pure mathematics and found total happiness.* Addison-Wesley, 1974.
9. Peter Koepke. Turing Computations on Ordinals. *The Bulletin of Symbolic Logic*, 11(3):377–397, 2005.
10. Peter Koepke and Martin Koerwien. Ordinal Computations. *Mathematical Structures in Computer Science.*, 16(5):867–884, 2006.
11. Mendelson. *Introduction to Mathematical Logic.* Wadsworth and Brooks/Cole Advanced Books of Software.
12. P.G. Odifreddi. *Classical Recursion Theory.* Elsevier, 1989.
13. Jacques Patarin. *Logique Mathématique et Théorie des Ensembles.* Polycopié de cours publié par l'École Centrale de Paris puis l'Université de Versailles-Saint-Quentin, 1991.
14. Jacques Patarin. Transfinite Cryptography. *HyperNet 2010, Tokyo 2010*, Also on ePrintArchive: Report 2010/001.
15. Apostolos Syropoulos. *Hypercomputation.* Springer-Verlag, 2008.
16. Philip D. Welch. Turing Unbound: Transfinite Computation. In *CIE '2007*, volume 4497 of *Lecture Notes in Computer Science*, pages 768–780. Springer-Verlag, 2007.
17. David Woodruff and Marten van Dijk. Cryptology in an Unbounded Model. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT '2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 149–164. Springer-Verlag, 2002.