Some Transfinite Generalizations of Gödel's Incompleteness Theorem

February 21, 2012

Jacques Patarin

UNIVERSITE DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

Agenda

1. Introduction

Gödel famous incompleteness theorem was first presented on October 7, 1930, at the first international conference of mathematic philosophy, at Königsberg.

This result of 1930 can be seen as a limitation result of usual computing theory: it does not exist a (finite) software that take as input a formula of order one on the integers and able to give as output (after a finite number of computations and with always a right answer) if this formula is true or false.

Classical computations

- In 1930 no real computer exited yet, but the mathematical analysis of the functions that can be effectively computed with (finite) software (i.e. “recursive functions”) had began.

- Gödel was also studying set of axioms such that there was a (effective, finite, recursive) computing way to know if a given formula was a member of these axioms or not.

The set of all true formulas on the integers

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

What we will do

In this work we will study what happens when use “transfinite softwares”, i.e. software that can be run on “transfinite computers”, and this means generalized computers that can perform α classical computations and use α bits of memory, where α is a fixed infinite cardinal. (For example $\alpha = \aleph_0$).

These transfinite computers are able to perform more things than classical computers, but, in another way, we can ask about their possibilities more “transfinite questions” that can be seen as generalizations of computations questions.

In fact, as we will see, it is possible to generalize almost all the classical results of limitation of the computation theory with this framework.

Previous work (1/2)

Such generalization is not totally new.

In [5] = [Patrick Grim, The Incomplete Universe, Totality Knowledge and Truth, MIT, 1991] and in some references mentioned in [5] , some problems linked with “Totality, Knowledge and Truth”, and “Incompleteness” are mentioned, and in [5] it is clearly explained that the fact that some limitation results can be generalized beyond the classical theory of computation is known since many years.

It seems however that an explicit description of the main limitation theorems in our framework of “transfinite computers” has not been done yet. In [5] for example the main subject is the problem of Totality of Knowledge and not α calculability where α is any cardinal.

Previous work (2/2)

As pointed out by an anonymous referee of WTCS2012, our transfinite computing model is in fact similar to admissible recursion on cardinals (which is equivalent to running ordinal Turing machines).

Admissible recursion has been well_developped since the 60s in work of Platek (1966), Kripke (1964) ans Sacks (cf Odifreddi, Classical Recursion Theory, 1989, p, 443).

2. Our transfinite computing model

Readers familiar with Ordinal Turing Machines (OTM), with tapes whose cells are indexed by ordinals, as described in [Peter Koepke, Turing Computations on Ordinals, The Bulletin of Symbolic Logic, 11(3):377- 397, 2005] can just not read this section 2.

The general idea is to follow a generalization of the Church's Thesis: as soon as a computation will be clearly feasible with $\leq \alpha$ bits of memory and $\leq \alpha$ computations, we will include it in the model.

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 generalized word, it means that when any set of operations has been performed, we know precisely what is the next operation to be performed.

The GOTO operation is an operation of the form (if $X = k$) then GOTO β where β is an ordinal. Here X and k are variables of α bits.

Remark on the memory

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 an infinity of times, without being fixed since then at 0 or 1.

However, it is possible to prove that if this value “not fixed” is changed with 0 (or 1), the infinite model of computation will be same (i.e. we will be able to compute the same functions), but the model is then maybe slightly less natural. (A variable B can be at 111…1… with an infinity of 1 if and only if a bit b has changed an infinity of time its value).

3. α–Recursive sets, α-Recursively enumerable sets

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

- **Definition 2** We will 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 ℝ of all the real numbers, or with [0, 1] for example.

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

- **Definition 4** By definition we will say that A is **α recursively enumerable** if and only if it exist at least one α-software P such that when we give $n \in I_\alpha$ as input of P:
- If $n \in A$ then P will be able to answer $n \in A$ after at most α operations.
- 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 $I_\alpha \to I_\alpha$. By definition, we will say that:
- F is α recursive $\Leftrightarrow$ it exist 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), we see that it exist some applications that are neither α recursive nor α recursively enumerable.

- **Definition 6**

- We will denote by $I_{\alpha \text{ limit}} = \cup \; I_\beta$ for all $\beta < \alpha$

And similarly we will define limit α softwares.

Note that limit $\aleph_0$ sottware are just the classical softwares.

4. Generalization of « recursively enumerable and not recursive sets »

α -Code of a α-software

We can associate very easily and injectively to each α–software T an element of I_α, named his α–code, and denoted $\ulcorner T \urcorner$.

Here by « easily » we mean that there exist α–softwares that take $\ulcorner T \urcorner$ as input, and then can find (and execute if needed) the sequence of α instructions of T.

Software result

If B is an α -software and x an element of , 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.

Software P

We can notice that there exist an α–software P which, when it is given $x \in I_\alpha$ as input:

1. « Find » the α–software X such that $\ulcorner X \urcorner = x$ if such α–software exist.
2. Execute the same instructions that X would execute with x as input.
   Thus we have:

$\forall x \in I_\alpha$, if there exists a α–software X such that $\ulcorner X \urcorner = x$, then $P(x) = X(x)$.

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) (when there exists X a α–software with code x).

Let $A = \{x \in I_{\alpha}$, such that P(x) is computed in $\leq \alpha$ computations$\}$

1. Since A is defined by the α–software P, A is **α recursively enumerable**.

2. If we assume that A is α recursive, let q be the code of a α–software Q such that:

$x \notin A \Leftrightarrow Q(x)$ is computed in $\leq \alpha$ computations

Then:

$q \in A \Leftrightarrow P(q)$ is computed in $\leq \alpha$ computations (by definition of A)

$q \in A \Leftrightarrow Q(q)$ is computed in $\leq \alpha$ computations (by definition of P)

$q \in A \Leftrightarrow q \notin A$ (by definition of Q)

This is not possible.

Thus A is **not α recursive**.

5. The decision problem

Theorem 2

There is no general algorithm, programmable with α–software, which could, using always $\leq \alpha$ computations, asserts if 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 where A is the set defined in theorem 1 above.

Since A is not α recursive, there exist no α software which, when applied to one of these propositions $n \in A$ can assert using $\leq \alpha$ computations if this proposition is true or false.

Remarks

1. We can also say that some properties that are true on chains of α bits are lost if we are limited to α computations and α bits of memory, for any infinite cardinal α.

2. These mathematical properties can be written with quantifiers $\forall$, $\exists$, the usual logic symbols and the usual operators +, x, and with $\leq \alpha$ elementary finite formulas. We then get a generalization of Gödel's incompleteness theorem (we just have to write the computations of α-software with such α-formulas, which are generalizations of order 1 classical formulas with α characters).

6. The halting problem

Theorem 3

There exists no α–software which can say with $\leq \alpha$ computations if a α–software will stop or not in $\leq \alpha$ operations.

Proof

If such α–software existed, then we could use it to write a α–software which, when it receive $x \in I_\alpha$ as input could say in $\leq \alpha$ operations if P (x) is computed after $\leq \alpha$ operations or not. But A is not α recursive, thus such a α–software do not exist.

7. The fixed point theorem on α-softwares

Notation

Let z, x, y $\in I_\alpha$ such that there exists a α–software Z whose code z has two entries: x and y.

We denote z[x, y] the output of the software Z on the 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 application α-recursive with two variables $s(x, y)$ such that:

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

Proof : cf paper

Theorem 6 (Fixed point theorem on α–softwares)

For all α–recursive application h there is an element $e \in I_\alpha$ such that:

$\forall\ x \in I_\alpha$, $e[x] = h(e)[x]$

Proof: cf paper

This means that if h is any α -recursive application, there exist always a α–software with code e and a α–software with code h(e) which on any input $x \in I_\alpha$ give the same output.

8. Rice theorem on α-softwares

Definition

We define « **α-recursive semi-functions** », any function f from D_f to I_α, where $D_f \subset I_\alpha$, such that there exists α–software which computes f (x) when it is given the 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 set of α-recursive semi-functions, different from the set of all these functions. Then:

A = {$n \in I_\alpha$, such that n is the code of a α-recursive semi-function of F} is not recursive.

Proof: cf paper

Applications of Rice theorem

This generalized Rice theorem shows that there exists no α–software to know:

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

Etc.

Therefore we see that this generalized Rice Theorem shows that the problem of a α–software « debugging », or the understanding of what a α–software is doing, generally uses more than α computations.

9. Some « philosophical » comments

- 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.

- The “ontological” definition of God (13th century) is : “God is the most powerful existing being that can be imagined without any contraction”. Is this definition in contradiction with our results ?

- In fact, I also do not take our mathematical arguments very seriously against a religious definition of God, 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.

Some possibilities to avoid the logical limitations ?

- 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.

- 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.

- Maybe God can access truth without computing.

- 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.

- You have to choose between universality and the possibility to create new sets from previous ones. If you choose universality many axioms of the usual set theory do not applies anymore.

10. Transfinite one way functions

It is not always easy to generalise results from classical computing theory to transfinite computing theory.

For example for public key cryptography the properties may be very different. One of the raison is the fact that we have billions of simple candidates to be one way functions on classical computing theory.

But so far I have found no candidate for transfinite computing theory. All the known simple candidate fail to have simple transfinite generalisations.

11. Conclusion

We have seen that most of the logic limitation results of the classical theory of computation can be generalized if we have devices able to perform α computations and use α bits of memory, where α is a given fixed cardinal.

It is expected that some other limitations results can also be generalized this way. This can be the subject of further work.