On Turing's Proof of The Undecidability of The Halting Problem

by Ulisses Ferreira

Abstract *- Since 1936, it has been well known that, given an arbitrary Turing machine and its input data, no Turing machine can decide in a general fashion whether the corresponding computation halts or not. However, in this article, it is shown that the formalism in that Turing's deduction contains some fundamental mistake. That is, this article does not prove that the halting problem is decidable but it may be seen as one step towards the decidability of the same problem.*

1 Introduction

In the present article, it is shown that the halting problem is possibly solvable, up to that Turing's proof[†].

It has been well known that the halting problem is unsolvable, and that it was originally demonstrated by Alan Turing himself, in both 1936 and 1937, in one of his remarkable articles[6]. If the problem is unsolvable for Turing machines, the proposition is often generalized. The converse can also be used.

The notion of composition of Turing machines is necessary for the halting problem. A definition is provided first. Since the Church-Turing thesis[2], it can be seen that Turing machines can be arranged in compositions. There is the notion of Tm composition, and the following is provided:

Definition 1 (Turing machine Composition) Let M be a Tm (possibly the Universal Turing machine) which interprets any Tm, and let $\{M_1, M_2, ..., M_n\}$ be a sequence $\mathcal{L}$ of occurrences of Turing machines where the codomain of one occurrence of Tm in the sequence $\mathcal{L}$ is the domain of its successive occurrence in the sequence $\mathcal{L}$. Furthermore, let X be the encoding of an arbitrary natural number in the domain of the first element of sequence $\mathcal{L}$, (M_1), where all domains in the same sequence are subsets of $\mathbb{N}$. A *Composition of Turing machines* over $\mathcal{L}$, for which the suggested notation is $M(M_n[...M_2[M_1[X]]...])$, is essentially the relative positions over the tape where the encoded machines rest (M and X indicate the context of the composition application but they are not part of such a composition). Those places on the tape are necessarily and previously set. Thus, $\mathcal{L}$ is placed on the tape forming a composition. Furthermore, M (or one occurrence of M, in case some other occurrence is on the tape as part of $\{M_1, M_2, ..., M_n\}$) is outside the tape, in such a way that, as a parallel between any Tm composition and Turing-computable function composition, the computation of $M(M_n[...M_2[M_1[X]]...])$ corresponds to the computation of another and single Tm, which in turn can also be referred to as *Turing machine composition*, C, given X, producing the respective result of $C(X)$ on the tape where it has been assumed that $C(X) = M(M_n[...M_2[M_1[X]]...])$, as the encoded resulting value v, $v \in \mathbb{N}$, in accordance with the Turing machine definition. Furthermore, for every $i \in \mathbb{N}$, as well as Tm M_i in the composition and hence in the sequence $\mathcal{L}$, M_i starts computing if for every j and M_j, $1 \leq j < i$, M_j results in a value in its codomain, necessarily in a subset of $\mathbb{N}$. Finally,

[†]He introduced, in a somewhat informal way for today, the claim together with his proof in 1936 and published the same article in 1937. At least if any logician or computer scientist have a look at his original reference[6] as well as more recent ones, he or she can see that the other authors, instead of Turing himself, have stressed the so called the halting problem in computer science.

for every $i \in \mathbb{N}$ as well as the corresponding $Tm\ M_i$, M_i does not produce a result without starting computing. Otherwise, the composition $M(M_n[...M_2[M_1[X]]...])$ is being assumed not to give any response if M or some involved machine in the Tm composition application does not give a response. Finally, the Turing machine composition gives a response if the execution of all involved Turing machines halt. □

As notation, for the computation of some composition, it is suggested to place $\uparrow$ as a prefix. For instance, $\uparrow\mathcal{H}(M[X])$ refers to the computation of $\mathcal{H}(M[X])$ at moment. Turing and others did not use such different notations, i.e. they did not distinguish a sequence of Turing machines and the corresponding computation.

At this point, the present author demonstrates that the proof of the undecidability of the halting problem, which used to be thought to be an application of the Cantor diagonal process, contains a fundamental mistake. In order to point out Turing's mistake, his proof is in books more or less as the following:

Theorem 1 *There does not exist any Turing machine $\mathcal{H}$ such that, for every Turing machine M and any input data X to M, both placed on the tape in a given representation and order, $\mathcal{H}$ can decide whether the computation of M given X halts or not.*

Let this proof be referred to as Turing's proof. Here the Church-Turing thesis is assumed for simplifying the proof. Assume that $\mathcal{H}$ is a program that decides whether (or not) M halts given X as input. In this way, given input X, it is previously arranged that $\mathcal{H}$ gives **true** if M halts, and gives **false** if M does not halt. Then, Turing defines another $Tm\ T$ that can be represented here in the following alternative form:

$$T(Y) \stackrel{def}{=} \textbf{if } \mathcal{H}(Y,Y) = \textbf{true } \textbf{ then } \perp \textbf{ else } \textbf{true};$$

where $\perp$ here represents the absence of response from a Tm as usual, that is, the absence of response from the Tm (T in the present case) by a programmable set of tuples that form a thread of computation which in its turn loops forever. Then, the logician who proves can make the following question: does the computation of $T(T)$ halt? At this point, one obtains the following representation of $T(T)$:

$$T(T) \equiv \textbf{if } \mathcal{H}(T,T) = \textbf{true } \textbf{ then } \perp \textbf{ else } \textbf{true};$$

and the answer of the question is the following: The application $T(T)$ halts (executing tuples that correspond to **else true** as above) if and only if the $\mathcal{H}(T,T)$ call results in **false**, and this is contradictory. Furthermore, $T(T)$ does not halt (executing $\perp$) if and only if the $\mathcal{H}(T,T)$ call results in **true**, and this is also contradictory. From these contradictions, it can be said that the assumption made is invalid, i.e. that the $Tm\ \mathcal{H}$ does not exist. □

While there are not many books on the theoretical computer science, [1, 4, 5, 3] are some examples. However, there is one important mistake in the above proof. In advance, the fact that the contradiction is found does not necessarily imply that the $\mathcal{H}$ algorithm does not exist. As it will be described, the evaluations of compositions can be different.

The thing is that $\mathcal{H}(Y,Y)$ apparently receives instances such as $\mathcal{H}(T,T)$ which, in a deeper sense, is not sufficient for universally representing this problem. In advance, and in accordance with T definition, it is well known that T is defined to be an algorithm based on $\mathcal{H}$ no matter how it had been implemented (say the latter had been implemented by someone else), which has characteristics of an interpreter or so, as the present author may well assume and, because of this, T receives a parameter (a variable representing one piece of input data) wherever T appears in this proof to make a computation.

In other words, as it was assumed that $\mathcal{H}$ is a program which makes dynamic analysis of another program, hence, by inspection, it follows that T also makes dynamic analysis of another program. And finally, a question is: what is analyzed in $\mathcal{H}(T,T)$? In the lack of an object that is dynamically analyzed, which, in contrast, cannot make dynamic analysis, the expression $\mathcal{H}(T,T)$ fails to represent correctly the application of an interpreter, and hence Alan Turing's proof fails to represent completely the problem.

More than this, it is known that the unsolvability of the halting problem is a universal claim (an existential claim negated). Therefore, his deduction in the original form does not prove his claim.

What halts is any program and data that, *together*, form computation. That is, the logician or computer scientist would need to represent all programs and data that would form any computation, not simply programs and data. In this way, Alan Turing would have previously established some representation of the empty input data since some programs do not require any data. Here, ϵ can be set for denoting empty data, while X donotes the representation of the value x on the tape. Thus, the application $\mathcal{H}(P, \epsilon)$ results in the *true* value iff P without input data halts. On the other hand, for some input $x \in \mathbb{N}$ where $x \neq \epsilon$, the application $\mathcal{H}(P, X)$ results in the *true* value iff P with input data X halts. However, the known proof of the undecidability of the halting problem does not represent both alternative possibilities. The proof of this is that he uses a program in place of data, and the program cannot be empty as universal.

Assume that $\mathcal{H}$ is the program to solve the halting problem, and further assume that $\mathcal{H}$ is an interpreter or so, and not a program that makes static analysis only. In this way, while a parser P can make syntax analysis on its source representation only (as an example of notation, $P(\ \langle P \rangle\ )$, where P denotes a program and $\langle P \rangle$ denotes its representation), $\mathcal{H}$ can also interpret itself. However, in the most inward level of composition, there is some non-interpretable piece of data (although data is not part of the composition, strictly speaking) or, alternatively, a program that does not require any input data. An example of representation of the first case can be $\mathcal{H}(\mathcal{H}(\mathcal{H}(P(X))))$ for some $\mathcal{H}$ unary, where P denotes a program, $P \neq \mathcal{H}$, and X denotes the possible input data where $X \neq \varepsilon$. An example of representation of the second case can be $\mathcal{H}(\mathcal{H}(Q))$ where Q denotes a program[†] that, by its definition, does not require any input data. The other example of the second case is obtainable from the first example by making $X = \varepsilon$. Therefore, in the lack of representation of both cases, the original proof of the unsolvability of the halting problem is incomplete. Therefore, what that known proof of the undecidability of the halting problem now seems to correctly state is that the halting problem solver is in a sense an interpreter, i.e. a program that makes analysis of computation itself, not simply a program that receives a program P and its input data X, and makes static analysis on P. More precisely, that proof seems to state that the possible halting problem solver has to have characteristics of an interpreter.

[Proof] Now and briefly, one referred to mistake of Turing's is introduced, as if the present author were playing a game together with Turing. From the following Turing's definition

$$T(Y) \stackrel{def}{=} \textbf{if } \mathcal{H}(Y, Y) = \textbf{true } \textbf{ then } \perp \textbf{ else true};$$

and $T(T)$, in the proof of the undecidability of the halting problem, as above stated, one obtains the following entailment for every program $\mathcal{H}$:

$$T(T) \ \Rightarrow\ \textbf{if } \mathcal{H}(T, T) = \textbf{true } \textbf{ then } \perp \textbf{ else true};$$

and now, as $\mathcal{H}$ is universally quantified, in a game against Alan Turing, the present author can state that $\mathcal{H}$ is an interpreter (or a kind of simulator of computation), together with its characteristics. Furthermore, regarding the call $\mathcal{H}(T, T)$ written by Turing, there are some general forms of evaluation, from which can be chosen to be part of $\mathcal{H}$:

- *Strict evaluation*: The machine evaluates the parameter first. Then it executes the sub-program or function. For instance, $sqrt(x + 1)$ where $x + 1$ is evaluated first and then the square root is calculated. With one or two exceptions, functional programming languages are divided in two classes, strict and lazy, and so all functions of the same programming language is either strict or lazy. However, this classification is conceptual and can be applied at the level of parameter.

[†]Notice that the first mathematician who used both notations, i.e. unary and binary, was Turing, and that, as a consequence, this article has to accordingly use both notations in such places, for permitting that proof of his.

- *Lazy evaluation*: The machine evaluates the parameter at most once. In the present Turing's proof above, the evaluation of one parameter of $\mathcal{H}(T,T)$ is strict as stated. Further, since the other parameter might not be used, the evaluation of the latter parameter can be lazy. The following item is more specific and suitable for what is needed.

- *Interpreter mode*: Since $\mathcal{H}$ can be assumed as having been specified and implemented by someone else, it is set that the first parameter is evaluated unconditionally, and because, in the proof, it is simply T, there is nothing interesting to do there other than resulting in T. That is, it is known that the parameter T in Turing's expression $T(T)$ does not need to be evaluated because T is a simple expression. Then, given $\mathcal{H}(T,T)$, the machine evaluates $\mathcal{H}$ first, which in its turn executes each step of the first parameter T, and this is what really happens as the present author assumed that $\mathcal{H}$ was an interpreter or so. In this way, depending on the first parameter of $\mathcal{H}$, i.e. the program interpreted by $\mathcal{H}$, the second parameter of $\mathcal{H}$ is either evaluated or not.

Therefore, whenever T is applied, $\mathcal{H}$ can be executed unconditionally, according to Turing's settings and these definitions over $\mathcal{H}$. In particular, whenever $T(T)$ is applied, since this parameter T is an expression that results in itself, wherever $T(T)$ occurs, $\mathcal{H}$ in $\mathcal{H}(T,T)$ is executed first.

Further, as $\mathcal{H}$ is assumed to be an interpreter or so, it has some property S of programs that have characteristics of interpreters or so, namely to make dynamic analysis of programs, among other characteristics. Since Turing plays with the universal quantifier, a person who specifies $\mathcal{H}$ can set that every program with property S requires a program (typically, the program that is interpreted) as a parameter.

By T definition, $\mathcal{H}$ not only has property S but also either T or $\mathcal{H}$ is executed unconditionally first. Thus, by T definition, it follows that T also has the property S. So $\mathcal{H}$ can be executed first and all programs in $\mathcal{H}(T,T)$ now have property S and, on the other hand, the computation of $\mathcal{H}(T,T)$ is in accordance with the following: $\mathcal{H}$ interprets the steps of the first occurrence of T; Since T has property S, the first occurrence of T interprets the steps of the second occurrence of T; The second occurrence of T does not have a program to interpret.

Therefore, in the lack of an occurrence of program to be interpreted (in accordance with the present author's settings, in his own right), Alan Turing's correct deduction cannot prove the unsolvability of the halting problem in a universal way. Notice that, although T is a Turing's definition, $\mathcal{H}$ is assumed to belong to others and, in this way, it can be unconditionally executed as soon as T is applied.

Finally, Turing's deduction was correct, only failed in the representation. □

References

[1] John E. Hopcroft, Jeffrey D. Ullman, and Rajeev Motwani. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[2] Stephen C. Kleene. *Introduction of Metamathematics*. D. Van Nostrand, Princeton, 1952.

[3] Harry Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Inc., second edition, September 1997.

[4] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1995. Reprinted with corrections.

[5] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[6] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936. (also in volume 43 (1937) pp. 544-546 with corrections).