Computation Is Not Conceptually Function Application

Ulisses Ferreira
e-mail: u.ferreira@philofcs.com

36 Pirapora, 41770-220, Salvador BA Brazil

Abstract

Since Turing's work, there have been attempts to refute Church-Turing thesis by trying to discover any effectively calculable partial recursive function that does not correspond to any Turing-calculable partial recursive function. Here, I do not only aim at pointing out an effect in the Turing machine model that has to do with Church-Turing thesis but also to fix it. Nonetheless, before having fixed it, the present author discovered a significant example which seems that has challenged it, and this article shows the example in question.

1 Introduction

Since 1930's, there has been very significant work into foundations of computer science, in theory of computation[9], category theory[8, 15] as well as in recursive function[1] theory, functional programming[18] and other theoretical subjects. Since Alan Turing, we can make references to many good researchers, but, to date, nobody seems to have observed unexpected effects in computations of compositions of Turing machines on the tape. Perhaps those good researchers observed what I have called unexpected effects whereas I

[1]The standard use of the term *recursive function* includes the notion of function that does not explicitly contain the operation called recursion, as presented and used in the recursive function theory. I use both terms, i.e. function and recursive function, as referring to the same notion. Accordingly, the common use for the term *partial function* includes the concept and semantics of *total function*, but since not all functions in the present article are necessarily total, I simplify the language using both terms, function and partial function, as referring to the same notion. Thus, in this article, *functions* and *partial recursive functions* have the same meaning.

observed not only the unexpected effects but also the possible rôle of unexpected effects in one of the semantics of computation. Those compositions are defined in [3]. Briefly and informally, let F and G be two Turing machines, and X be some input. I shall show that, if a programmer wants to form a composition such as $F(G(X))$ with both machines on the tape, we shall have to observe the dynamic possibility of G changing F for another Turing machine, say F'. Comparing F and G in this context, while G in this context does not prove anything other than $G(X)$, a Turing machine must prove something more than $F(G(X))$, i.e. a Turing machine must prove that the G calculation does not affect the F calculation. In this paper, we shall see such details in a careful and more precise way. Variations on Universal Turing machines have been proposed[10] but not much work has been carried out in the only and traditional computability theory, which may cause the impression that the original computability theory is established. The fact is that, because of the parallel pioneering pieces of work by Church and Turing, we still have what is called Church-Turing thesis[20], or even called Church thesis[19].

In section 2 I present Turing machines from the operational standpoint[2]. In section 3 I give interpretation of other notions used in the current paper and, in section 4, I present my claims.

2 Turing machines

Perhaps the best definition that I have seen regarding non-deterministic Turing machine is in [11], and I reproduce it here with that notation, although I do not further use that notation:

Definition 1 *A Turing machine (TM) is an ordered system* $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *where* Q *is a finite set of states,* Σ *is the input alphabet,* Γ *is the tape alphabet,* $\Gamma \cap Q = \emptyset$ *and* $\Sigma \subset \Gamma$*,* $q_0 \in Q$ *is the initial state,* $B \in \Gamma - \Sigma$ *is the blank symbol,* $F \subseteq Q$ *is the set of final states and* δ *is the transition function,*

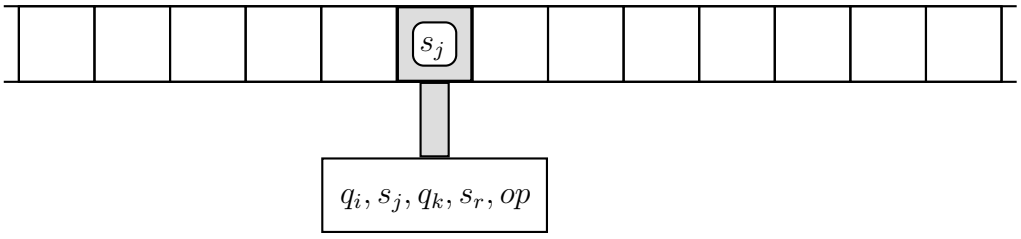
$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

□

[2]In the present article, I prefer to use the term *operational* instead of the term *intentional*, for the former suggests that the corresponding scope is the artificial computation. Intention is a psychological notion more complex than operation. Although such words are well established in the computer science community since philosophy of mathematics, those from the community can also gradually reserve the latter for future use.

By defining the codomain of δ as above, that is $\mathcal{P}(Q \times \Gamma \times \{L, R\})$, the machine TM may be non-deterministic. Furthermore, it can be shown that non-deterministic Turing machines are equivalent to deterministic Turing machines. For a deterministic version, which I use in this thesis, δ can be redefined as $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$. In this section, however, as a matter of convenience for the present purpose, I briefly define Turing machines in a slightly different way, and this definition has also been used by other authors such as [1, 2, 6, 7, 9, 12, 13, 14, 17, 21] as well as Turing. The two differences are that, here, a Turing machine program is a set of tuples (or a set of γ-transitions, as one may prefer to refer to) and there is one tape. As many of these simple transitions produce some effect on the tape, and this tape is shared by other programs, assuming Church-Turing thesis, what necessarily corresponds to a function is ultimately the whole sequence of Turing machines, including all machines that are interpreted, with the involved compositions if the Universal Turing machine is not outside the tape in order to guarantee that the computation is free from unexpected effects. Indeed, this is one of the results in this paper: some assumptions have to be added to this assertion.

Without changing the notion, a Turing machine (TM) has also been defined as a 6-tuple, $M = (Q, \Sigma, T, P, q_0, F)$ where Q is a finite set of states $\{q_0, q_1, ..., q_n\}$, Σ is the alphabet which is a finite set of symbols $\{s_0, s_1, ..., s_m\}$, where s_0 is the blank symbol that I represent with $\circ$, $T \subseteq \Sigma \setminus \{\circ\}$ is the set of input symbols, P is the Turing machine program, $q_0 \in Q$ is what has been called initial state, and $F \subseteq Q$ is a set of final states of this Turing machine. According to Alan Turing's analogy, a Turing machine is supplied with an infinite tape divided into squares and one write/read head. Each tape square contains one symbol in Σ.



The diagram for Turing machine, with its tape squares

Given $\mathcal{O} = \{L, R, S, H\}$, the program P is a set of nn transitions or γ-transition functions, for $0 \leq ii \leq nn, \gamma_{ii} : Q \times S \longrightarrow Q \times S \times \mathcal{O}$ to which has been referred as a set of 5-tuples of form (q_i, s_j, q_k, s_r, op), $op \in \mathcal{O}$, such

that each 5-tuple in P produces an effect that is described in the literature in the following way:

As usual and in accordance with Alan Turing's original analogy using a tape, the write/read head is initially on the leftmost non-blank symbol, which means the leftmost symbol of the input for the Turing machine. The write/read head writes and reads symbols on the tape as it moves along the tape, one square to the right or to the left, depending on the current state of M and the current symbol, i.e. on which the write/read head is. For the purposes of this discussion, without loss of generality, I can set that, in the initial state q_0 of M, the write/read head is on the leftmost non-blank symbol of the tape. Thus, the machine works in the following way: for every simple[3] step of calculation, for some 5-tuple in P, if the machine is in state q_i and the write/read head is on the square which contains some symbol s_j, the machine substitutes s_r for s_j in the same square, substitute q_k for the current state, and perform one of the following actions:

- if $op = L$, the machine moves the write/read head one square to the left.
- if $op = R$, the machine moves the write/read head one square to the right.
- if $op = S$, the machine does not move the write/read head.
- if $op = H$, the machine halts.

Just a short note on notation used in the present paper, from now on: given some Turing machine M as the first operand and some p which is represented here as a letter in the set of symbols $\{F, P, Q, T, \Sigma\}$ as the second operand, I define the meta-language infix operator $\star$ to denote a set from the Turing machine, i.e. $M \star p$ denotes the corresponding set in M. For instance, $M \star P$ refers to the program of M and $M \star Q$ refers to the states of M.

As in the definition 1, a Turing machine does not have to have a tape or a write/read head. Here, we can define the alphabet $\Sigma_2 \stackrel{def}{=} \Sigma \cup \{\odot\}$ and the language $\mathcal{T}$ in Σ_2^* where $\odot \notin \Sigma$ is the symbol that indicates that the next symbol on the right of $\odot$ is under the write/read head. Thus, the tape is merely a string in Σ_2^*. To keep the comparison, the symbol $\odot$ occurs only

[3]In this article, I use the term *simple* step for both Turing machines and effective computation, although we only need to go a little more in detail here, as I am describing Turing machines. In a modern sense of computation, a simple step can migrate an agent, for instance, from one country to another because such an atomic operation is well defined in some way. Therefore, nowadays that simplicity is subjective and not relevant, and, because of this, I sometimes simply use the term *step* instead.

once in the string. Those strings are infinite but only one finite part of them can contain non-blank symbols. Thus, so far the transition functions become: $0 \leq ij \leq nn,\ \gamma_{ij} : \Sigma_2^* \times Q \longrightarrow \Sigma_2^* \times Q$. Furthermore, because the state $q_i \in Q$ together with the symbol on the right of $\odot$ determine the transition function γ_{ij}, the function γ_{ij} can be defined as: for $\Sigma_3 = \Sigma_2 \cup Q, \Sigma_2 \cap Q = \emptyset :\ \gamma_{ij} : \Sigma_3^* \longrightarrow \Sigma_3^*$. A non-encoded Turing machine can be seen as a grammar.

I shall explain that, on a shared tape, the computation by the γ-transitions of one encoded Turing machine may affect the γ-transitions of other encoded Turing machines. Once Turing machines are encoded and placed on a tape, they are still separate entities while now they share the same tape. Therefore, in this paper, I am not going to view Turing machines as a form of rewriting systems over strings, but, instead, from the operational standpoint, regarding compositions between Turing machines and so forth. Furthermore, regarding representation, I am going to use one tape and one write/read head, as explained in many books on computability theory since Alan Turing's papers, among others. I understand that this analogy with a physical machine is necessarily sound and helps the explanation.

In this article, I introduce an example and observe one feature that is present in one notion and absent from another. I use the notation $M[X]$ to stand for the computation of a Turing machine M that inputs x, where X is the representation of x. To describe the computation of a Universal[4] Turing machine when simulating $M[X]$, I denote $U(M[X])$ instead of $U[M[X]]$. Accordingly, the functional composition $m(n(x))$ from two Turing machines, M and N, are denoted by $M(N[X])$. If there exists a Universal Turing machine interpreting this composition, that is $u(m(n(x)))$, I denote this situation by $U(M[N[X]])$. In this way, I use parentheses at the outermost level and brackets internally to make it clear that the former applying Turing machine is not represented on the tape, but instead outside the tape, while the latter is represented on the tape.

As notation for the computation of some composition, I make use of the up arrow symbol as a prefix. For instance, $\uparrow M(N[X])$ refers to the computation of $M(N[X])$ in the present piece of work.

Below, I define the Universal Turing machine with the characteristics that will be subsequently helpful in this article.

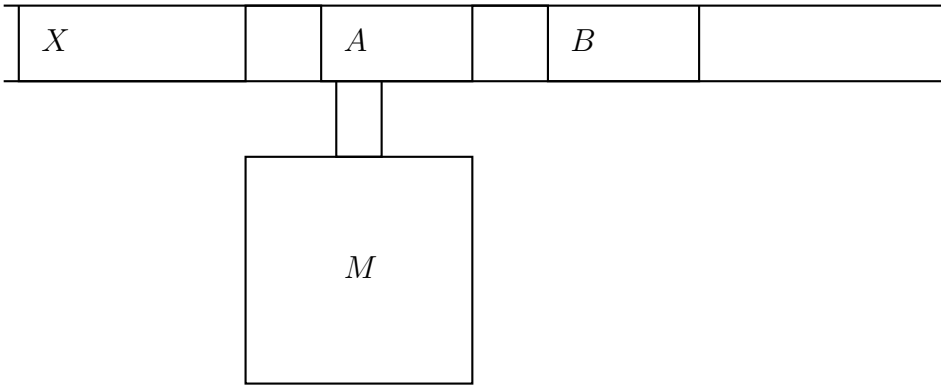
Definition 2 (Universal Turing machine) *Let U be a Turing machine.*

[4]In this article, I use both "a Universal Turing machine" and "the Universal Turing machine" as meaning the same. I use the former when I want to refer to a class of Universal Turing machines that universally interpret Turing machines, and use the latter when I want to stress the result, i.e. the interpretation itself. In both cases, my view is operational.

U is a Universal Turing machine if and only if, given any Turing machine $M : \mathbb{N} \longrightarrow \mathbb{N}$ *encoded and placed on the tape, M corresponding to the Turing-computable function* $m : \mathbb{N} \longrightarrow \mathbb{N}$*, and given any input* $X : \mathbb{N}$*, which is some representation of the natural number* $x \in \mathbb{N}$ *on the tape, U calculates the value* $m(x)$*.*

□

To help the reader to understand this article, notice that the notion of function exists if and only if the notion of composition exists as a property (amongst the others). In this case, one cannot talk about functions without considering compositions as one of the fundamental properties of any related theory, in particular, theory of computation. Furthermore, let $f : D_1 \longrightarrow D_2$ and $g : D_2 \longrightarrow D_3$ be two total functions. There are at least the same number of Turing machines that assign to the values in D_1 the values in D_3 by implementing the $g(f(x))$ result (by concatenating corresponding Turing machines or by any other means), than the number of Turing-computable functions with the same results by concatenating corresponding Turing machines, whether or not there is one such a Turing-computable function. Thus, this article shows that, more than that, depending on one hypothesis that I shall observe, there can be those compositions of the referred to Turing machines that yield values outside D_3. As an initial example of Turing machine composition, following a convention, the composition $M(B[A[X]])$ can be as in the following diagram:



A Turing Machine Composition

As a usual example of arrangement (among others), one would previously establish that the input of a Turing machine is always on the left of its en-

coding and that they are separated by precisely one square. Furthermore, after having finished interpreting a Turing machine, M clears the interpreted Turing machine as well as its input, and then places the output of this interpretation in the correct place before starting interpreting another Turing machine, and so forth. Finally, in this example, one can establish that every Turing machine in the composition has its reserved space on the tape on the left of its input, but this is only *a priori* operational convention and, as such, does not prevent unexpected effects. The important point is that M has to prevent unexpected effects between Turing machines.

In category theory, for example, there are objects, arrows, functors, natural transformations, monads and many other concepts. Concepts are used to define more sophisticated concepts, and the essence or basis is sets and functions, as well as properties. Although the concept of monad[16] can be used to justify mathematically input/output in functional programming, that idea is far from refuting the present work on unexpected effects on Turing machines. One of the reasons is that all those computable functions have signature $\mathbb{N} \longrightarrow \mathbb{N}$, sometimes represented with encoding the signature $\mathbb{N}^k \longrightarrow \mathbb{N}$ for some $k \in \mathbb{N}$. The Turing machine theory is relatively simple, and has been supported by programming. Likewise, the present refutation should be as such, using the same signature $\mathbb{N} \longrightarrow \mathbb{N}$ instead of more complex functional notions. Moreover, the meanings of the term "unexpected effect" are definitely not the same, with respect to functional programming. An essential difference between the concept of side-effects in input/output operations in functional programming and the term unexpected effect here is that, in the former case, the operation is programmed. In other words, there is control and intention. Here, unexpected effect during a computation is unpredictable from the point of view of the programs. Concisely, some unexpected effect appears during the computation (and possibly in books of computability theory) depending on the following:

1. The absolute positions of Turing machines on the tape;
2. Whether or not one Turing machine represented on the tape, by chance, affects another one represented on the tape;
3. Whether or not the Turing machine outside the tape avoids unexpected effects. Both alternatives indeed exist.

3 Some interpretations

In this section I present the interpretations of the notions used in my theorems. In a philosophical and insightful chapter [5] by Prof Galton, there is

a discussion on different interpretations of Church-Turing thesis, including different assertions made for the thesis.

- **Model of Computation -** In this paper, because I review a well-established model of computation only, it suffices to see computation as simply a sequence of (possibly infinite) simple steps that belong to some well-established model of computation. Additionally, computation is carried out at some place and takes time. I do not redefine, intuitively or formally, *model of computation* here. Instead, for comparisons, I assume that λ-calculus is a model of effectively calculable functions. As suggested, a computation may be empty.

- **Effective Calculation -** For any natural number k, a function $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ is *effectively* calculable if and only if there exists some finite procedure p represented in h, that is, a unique function $ph(p) = h$, and a unique number-theoretic partial function $g\colon \mathbb{N}^k \longrightarrow \mathbb{N}$, such that given $x \in \mathbb{N}^k$, h calculates the value $g(x)$ according to p.

  A more precise definition is the following: Let MC be the set of all models of effective computation, and P be the set of all simple steps regardless of the model. Let P^* denote the infinite set of all sequences of such simple steps, many of which are meaningless for they are steps from different models, and many of these sequences of steps are infinite, and let $p \in P^*$ denote a finite sequence of steps that form an effective procedure of some model $M \in MC$. Further, let $p[x]$ denote the computation of some procedure p given some value x, and $S \in P^*$ denote a (possibly infinite) sequence of simple steps carried out by the same computation. Let $S = p[x]$. Therefore, given the above notation, for any natural number $k > 0$, $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ is said to be an effectively calculable function if and only if there exists a number-theoretic function $g\colon \mathbb{N}^k \longrightarrow \mathbb{N}$, a function $f : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}$, and a unique function $\mu : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N} \times MC \times P^*$ and also $\mu(f, M, S) = h$ (possibly many to one) for each (f, M, S, h), that denotes h, and for every $M \in MC$ and any $x \in \mathbb{N}^k$, there exists a sequence of simple steps $S \in P^*$, $S = p[x]$, and finally, the calculation of $h(x)$ or $f(M, S, x)$ is in accordance with one of the following alternative cases:

  - If the value of x is defined in g, the calculation follows a finite sequence of simple steps S (halts), and both $h(x)$ and $f(M, S, x)$ must result in the value of $g(x)$.
  - If the value of x is not defined in g, a situation commonly and formally represented as $g(x) = \perp$, the calculation follows an infinite

sequence of simple steps, i.e. $|S| = \infty$ and the application $h(x)$ or $f(M, S, x)$ never halts.

Roughly, $f(M, S, x) = g(x) = h(x)$, where $x \in \mathbb{N}^k$, $M \in MC$ and $S \in P^*$. More formally and using first-order predicate logic, and a predicate ec that states whether a function is effectively calculable,

$$\forall k \in \mathbb{N}.\ \forall f : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}.\ ec(f) \equiv$$
$$\exists!(g : \mathbb{N}^k \longrightarrow \mathbb{N}).\ \forall(M \in MC).\ \exists(S \in P^*).\ \forall(x \in \mathbb{N}^k).\ f(M, S, x) = g(x)$$

where g is unique since the quantifier indicates.

To allow composition of functions I can repeat the parameter M for the same model and sequences of steps S_0 and S_1,
$f(M, S_1, f(M, S_0, x)) = g(g(x))$ holds here because $k = 1$. However, for $k > 1$, effectively calculable functions must accept and result in one encoding number, e.g. a Gödel number, and, by temporarily reducing k to 1, I do not loose generality. That is, every effectively calculable function must decode x before its calculation and, before resulting its final value, it must encode its result into a natural number. In the present paper, after these definitions I shall use only functions with $k = 1$. The next interpretation is a particular case of this one.

- **Turing Computability and Calculable Functions -**

  For some partial and number-theoretic function $g : \mathbb{N}^k \longrightarrow \mathbb{N}$, some numbering interpretation (some codification previously established) $\mathcal{N}$ from a symbolic representation, for example including Gödel numbers, a partial function $t : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}$ is a Turing-calculable (or simply computable, or calculable) function if and only if there exists a Turing machine T such that, given the representation $r(x)$ (according to $\mathcal{N}$) of the value $x \in \mathbb{N}^k$ and once $r(x)$ is placed at some established place in the tape for input by that machine, and given a sequence of simple steps $S \in P^*$, which is the program of T, the calculation of $g(x)$ by t is in accordance with one of the following cases:

  - if x is defined in g, the calculation of t halts in a finite number of simple steps S in a final state of T leaving the appropriate representation of $g(x)$, i.e. according to $\mathcal{N}$, at the established place of the tape for the result.
  - if x is undefined in g, either the calculation of t does not halt or it halts in a state $s \notin M \star F$.

Intuitively and in a somewhat informal way, a predicate that states whether f is a Turing-calculable function (asserted using the predicate formula $tcf(t)$) is defined as follows: for all $\mathcal{N} \in I$,

$$\forall(k \in \mathbb{N}).\ \forall(t\colon MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}).\ tcf(t) \equiv \exists!(g\colon \mathbb{N}^k \longrightarrow \mathbb{N}).\ \exists(S \in P^*).\ \forall(x \in \mathbb{N}^k).\ \mathcal{N} \models t(TM, S, x) = g(x)$$

where TM is constant which is a value in the domain MC, I denotes the set of all possible codifications, and g is naturally unique.

From now on, I shall not write $\mathcal{N}$ in the formulae for the purpose of simplification, as I have already stated that an interpretation always exists and I simply assume that it is constant.

Thus, I shall demonstrate in proposition 2 that Turing machines do not necessarily entail functions. I demonstrate that unexpected effects introduce a problem that has a logical aspect and a conceptual one.

3.1 Unexpected effects

We can view *unexpected effect* as being the effect of some operation that a Turing machine (or, more generally, a program) can perform that can possibly change the representation of data or Turing machine on the tape (or of data or program in a common memory device) and therefore its result, in such a way that the effective effect depends on where the representations of the Turing machines are placed on the tape, as well as secondary conditions. If they are placed in different blocks in different runs, the results from these occurrences of computations are possibly different. The notion of unexpected effect is particularly important in any interpretation of Turing machine by a Universal Turing machine, for the latter has to guarantee the absence of unexpected effects, as we shall see. One can formally define the notion of unexpected effect as follows:

More abstractly, there exist two instants, $t_0 \neq t_1$, of time such that

$$\frac{TM,\ M \vdash @' \cdot t_0[\uparrow M[X] = m]\ \wedge\ TM,\ M \vdash @' \cdot t_1[\uparrow M[X] = n]\ \wedge\ m \neq n}{TM,\ M \vdash se(\uparrow M[X])}$$

The above definition is in terms of proof (if the results from the same Turing machine are different, I prove that there exists some unexpected effect). Or alternatively as follows:

$$se(M[X]) \equiv @TM \cdot' [@M \cdot t_0[M(X)] \neq @M \cdot t_1[M(X)] \wedge t_0 \neq_t t_1]$$

where M is a Turing machine, $M(X)$ here denotes the same as $M[X]$ and both denote M running for some input x, TM is the Turing machine model of computation, $@' \cdot t[M[X]]$ is the result from computation $M[X]$ at time t and under some set interpretation of encoding results, and $se(M[X])$ is the predicate that states the existence of unexpected effects in a Turing machine application $M[X]$.

Notice that unexpected effect is not a pure-mathematical notion regardless of its importance in computer science. Furthermore, M might produce or receive unexpected effects. For the analysis in the next section, one can interpret that an unexpected effect indeed replaces one Turing machine by another one, while they are interpreted by a Turing machine that neither detects nor treats unexpected effects. From this perspective for unexpected effects, one alternatively interprets one unexpected effect as follows:

$$\begin{aligned} se(M[X]) \equiv @TM \cdot' [(\exists M, M')\ @s \cdot t_0[thereis(M)] \wedge @s \cdot t_1[thereis(M')] \\ \wedge M \neq M' \wedge M(x) \neq M'(x) \wedge t_0 <_t t_1] \end{aligned}$$

where s denotes a place on the tape, $thereis(M)$ is a predicate that denotes the existence of the Turing machine M. Notice that the above formula makes use of the closed-world assumption.

4 A Refuting Example

In the following theorems of this paper, I prefer to use both terms *computable* and *calculable* as meaning the same.

In my analysis, there exist two connections, namely, between Turing machines and Turing-computable functions, and between Turing-computable functions and effectively computable functions. I am at showing that the former one-to-one correspondence is broken, as well as their structural properties are not preserved because of the necessary notion of composition.

Example 1

Let U be a Universal Turing machine, and let G and H be two Turing machines that are placed on the tape. For my proofs, an example will suffice. Thus, as an example, H calculates double its input, which is encoded in binary and placed on the left of H, separated by, say, fifty blank symbols, initially. Thus, $H \star \Sigma = \{\circ, 0, 1\}$ and $H \star T = \{0, 1\}$. In this example, let $H \star P$ be

$(q_0, 0, q_2, \circ, R)$, $(q_0, 1, q_1, \circ, R)$,
$(q_2, 0, q_2, 0, R)$, $(q_2, 1, q_2, 1, R)$

$(q_2, \circ, q_4, 0, L)$, $(q_4, 0, q_4, 0, L)$, $(q_4, 1, q_4, 1, L)$,
$(q_4, \circ, q_5, 0, H)$,
$(q_1, 0, q_1, 0, R)$, $(q_1, 1, q_1, 1, R)$
$(q_1, \circ, q_3, 0, L)$, $(q_3, 0, q_3, 0, L)$, $(q_3, 1, q_3, 1, L)$,
$(q_3, \circ, q_5, 1, H)$.

Thus, $H \star Q = \{q_0, ..., q_5\}$, $n = 5$, $m = 2$, and $H \star F = \{q_5\}$.

Let $G = (H \star Q \cup \{q_{n+1}, ..., q_{n+3+w}\}, H \star \Sigma, H \star T, \Lambda \cup G \star P, q_0, H \star F \setminus \{q_0\})$ be defined as follows:

G moves the write/read head an arbitrary number w of squares to either left or right of $r(G)$, and, for some $s \in G \star \Sigma$, writes s on the tape. In this way, the Turing machine G is similar to H, except that G attempts to produce some unexpected effect on the tape. Without loss of generality, this can be done in the following way, assuming that I choose to move the write/read head to the right and that the write/read head is positioned at the leftmost square of $r(G)$ at q_0:

$\forall \gamma \in H \star P, \gamma \equiv (q_i, a, q_j, c, d) : i \neq 0 \wedge j \neq 0 \Rightarrow \gamma \in \Lambda$.
$\forall \gamma \in H \star P, \gamma \equiv (q_0, a, q_i, c, d) : \gamma \notin \Lambda \wedge (q_{n+1}, a, q_i, c, d) \in \Lambda$.
$\forall \gamma \in H \star P, \gamma \equiv (q_i, a, q_0, c, d) : \gamma \notin \Lambda \wedge (q_i, a, q_{n+1}, c, d) \in \Lambda$.
$q_0 \in H \star F \Rightarrow q_{n+1} \in G \star F$.
$(q_0, s_0, q_{n+2}, s_0, S) \in G \star P$.
$\forall s \in \Sigma \setminus \{s_0\} : (q_0, s, q_0, s, R) \in G \star P$.
For some arbitrary $w \in \mathbb{N}$:
$\forall i \in \mathbb{N}\ (i < w) : (q_{n+2+i}, s_0, q_{n+3+i}, s_0, R) \in G \star P$.
$(q_{n+2+w}, s_0, q_{n+3+w}, s_1, L) \in G \star P$.
$\forall i \in \mathbb{N}\ (i < w) : \forall j \in \mathbb{N}\ (1 \leq j \leq m) : (q_{n+3+i}, s_j, q_{n+3+w}, s_0, L) \in G \star P$.
$(q_{n+3+w}, s_0, q_{n+3+w}, s_0, L) \in G \star P$.
$\forall s \in \Sigma \setminus \{s_0\} : (q_{n+3+w}, s, q_{n+3+w}, s, L) \in G \star P$.
$(q_{n+3+w}, s_0, q_{n+1}, s_0, R) \in G \star P$.

Notice that, like H, G finally halts in s_5. That is, both $(q_4, \circ, q_5, 0, H)$ and $(q_3, \circ, q_5, 1, H)$ are in $G \star F$. Therefore, G is an algorithm.

Now, given $X : \mathbb{N}$, some Turing machine $F : P^* \times \mathbb{N} \longrightarrow \mathbb{N}$, and sequences of simple steps S and S_2, let $U(F[G[X]])$ be calculated: Suppose for the present example that F calculates the integer division modulus four of a number represented in binary digits (that is, F results in the two least significant digits). I define F as

$$F \star Q = \{q_0, q_{10}, q_{11}, q_{12}, q_{13}, q_{100}, q_{101}, q_{102}, q_{103}, q_{1000}, q_{1001}\}$$

and $F \star F = \{q_{1000}, q_{1001}\}$. Thus, $F \star P$ can be defined as follows:

$(q_0, 0, q_{10}, \circ, R),\ (q_0, 1, q_{11}, \circ, R),$
$(q_{10}, 0, q_{10}, 0, R),\ (q_{10}, 1, q_{11}, 1, R),\ (q_{10}, \circ, q_{100}, \circ, L),$
$(q_{11}, 0, q_{12}, 0, R),\ (q_{11}, 1, q_{13}, 1, R),\ (q_{11}, \circ, q_{101}, \circ, L),$
$(q_{12}, 0, q_{10}, 0, R),\ (q_{12}, 1, q_{11}, 1, R),\ (q_{12}, \circ, q_{102}, \circ, L),$
$(q_{13}, 0, q_{12}, 0, R),\ (q_{13}, 1, q_{13}, 1, R),\ (q_{13}, \circ, q_{103}, \circ, L),$
$(q_{100}, 0, q_{100}, \circ, L),\ (q_{100}, 1, q_{100}, \circ, L),\ (q_{100}, \circ, q_{1000}, 0, R),$
$(q_{101}, 0, q_{101}, \circ, L),\ (q_{101}, 1, q_{101}, \circ, L),\ (q_{101}, \circ, q_{1001}, 0, R),$
$(q_{102}, 0, q_{102}, \circ, L),\ (q_{102}, 1, q_{102}, \circ, L),\ (q_{102}, \circ, q_{1000}, 1, R),$
$(q_{103}, 0, q_{103}, \circ, L),\ (q_{103}, 1, q_{103}, \circ, L),\ (q_{103}, \circ, q_{1001}, 1, R),$
$(q_{1000}, \circ, q_{1000}, 0, H),\ (q_{1001}, \circ, q_{1001}, 1, H).$

and then, supposing $x = 93$, we obtain the following situation in q_0:

$$\text{tape starts here} \longrightarrow \underset{\diamond}{|}1011101 \circ ...r(G) \circ ...r(F) \circ ...$$

where $\diamond$ is the write/read head.

Because some simple steps of calculation of G might modify the representation of any Turing machine placed on the tape, including of F, we could obtain $U(F[G[X]]) \neq U(F[H[X]])$ from the calculation. The programmer who writes F does not have prior knowledge on G nor H. That is, G might change the representation of F if U allowed this. From the alternative view for unexpected effects, a computation could start as $U(F[G[X]])$ and finished resulting in $U(F'[G[X]])$ since G might change the Turing machine F in such a way that it would become F', if U allowed G to do so.

□

Definition 3 *For this article, let $k \in \mathbb{N}$, $k \geq 0$, $X : \mathbb{N}$ be some input, and $k+1$ Turing machines $F_k : \mathbb{N} \longrightarrow \mathbb{N}$. For any $k > 0$, a ($k-$level) Turing-machine composition is a composition of $k+1$ Turing machines $F_k[F_{k-1}[...[F_0[X]]...]]$. For any $0 < i \leq k$, F_i does not read or manipulate any Turing machine other than F_{i-1}.*

Lemma 1 (Universal Interpretation) *For any $k \in \mathbb{N}$, for any representation $X : \mathbb{N}$ on the tape, and for any Turing machines $F_0, F_1, ..., F_k$, let $F_k[F_{k-1}[...[F_0[X]]\ ...]]$ be a k-level Turing-machine composition. Then, the Universal Turing machine is capable of reading the Turing machines $F_0, F_1, ..., F_k$.*

[Proof] To calculate any $U(F_k[F_{k-1}[...[F_0[X]]...]])$, U interprets the operations of some of the involved Turing machines, i.e. some of $F_0, F_1, ..., F_k$, by following either lazy or strict evaluation.

□

Lemma 2 *Let $X : \mathbb{N}$, U be the Universal Turing machine, $M_0, ..., M_k : \mathbb{N} \longrightarrow \mathbb{N}$ be $k+1$ Turing machines, and $U(M_k[M_{k-1}[...[M_0[X]]...]])$ be a k-level Turing-machine composition where $k > 0$. There exists a non-empty set of transition functions in the Universal Turing machine that guarantees absence of any unexpected effect at any level $i \leq k$ in Turing-machine compositions.*

By example 1, a Universal Turing machine has to get round the problem of unexpected effects. In this article, the way is not important, but it may be done by manipulating the tape configuration whenever the calculation of a Turing machine tries to modify another machine on the tape. That is, for all sequences of steps, U must always guarantee $\forall F, G, H : \mathbb{N} \longrightarrow \mathbb{N}, \forall X : \mathbb{N}, U(F[G[X]]) = U(F[H[X]])$. Therefore, since the programmable part of a Turing machine is in its set of transitions, there exists a non-empty set of transitions $\mathcal{S} \subset U \star P$ that can solve this problem of unexpected effects.

□

Theorem 1 *The class of Turing machines is not isomorphic to the class of effectively computable partial recursive functions. Furthermore, neither the former is necessarily equivalent to the latter, e.g. two Turing machines can correspond to the same function, nor all structural properties of the class of Turing machines correspond to the structural properties of the class of effectively computable functions with respect to the notion of composition.*

[Proof] By lemma 2, there exists a non-empty set of transition functions $\mathcal{S} \subset U \star P$ that can solve the problem of unexpected effects. Now, let $U(U[G[X]])$ be calculated, from which the reader obtains the following situation in $U \star q_0$ and in $G \star q_0$:

tape starts here $\longrightarrow |1011101 \circ ...r(G) \circ ...r(U) \circ ...$
$\diamond$

and the final situation in $G \star F$ containing the double value, 186, is

tape starts here $\longrightarrow |10111010 \circ ...r(G) \circ ...r(U) \circ ...$
$\diamond$

although solution $\mathcal{S}$ might move the absolute positions of $r(G)$ and $r(U)$, and hence changing the tape configuration. Thus, the computation of $G(X)$ is represented as follows:

$$
\begin{array}{llllll}
q_0 1011101\circ & \longrightarrow & \circ q_1 011101\circ & \longrightarrow & \circ 0 q_1 11101\circ & \longrightarrow \\
\circ 01 q_1 1101\circ & \longrightarrow & \circ 011 q_1 101\circ & \longrightarrow & \circ 0111 q_1 01\circ & \longrightarrow \\
\circ 01110 q_1 1\circ & \longrightarrow & \circ 011101 q_1\circ & \longrightarrow & \circ 01110 q_3 10 & \longrightarrow \\
\circ 0111 q_3 010\circ & \longrightarrow & \circ 011 q_3 1010\circ & \longrightarrow & \circ 01 q_3 11010\circ & \longrightarrow \\
\circ 0 q_3 111010\circ & \longrightarrow & \circ q_3 0111010\circ & \longrightarrow & q_3 \circ 0111010\circ & \longrightarrow \\
q_5 10111010 \circ . & & & & &
\end{array}
$$

Assuming that there is no unexpected effects in the above computation. Then let two Turing machines, U and V, exist such that, except for the possibility of unexpected effects, U and V produce the same output: The only difference is that U contains $\mathcal{S}$ and calculates $U(U[G[X]])$, and V does not contain $\mathcal{S}$ and might calculate $V(V[G[X]])$ or $V(U[G[X]])$. As a possible example, V may sometimes calculate $U(U[G[X]])$ and sometimes not, depending on the physical places where V and G rest on the tape. Assuming that the class of Turing machines necessarily corresponds to the class of effectively computable functions, for later contradiction (although my Example 1 above clearly applies to any model based on functions), I can choose λ-calculus, defined by Church himself, as a functional model of effective calculability, denoted here by $\lambda{-}calculus \in MC$. Clearly, a simple case by case analysis demonstrates that parameters in λ-calculi cannot modify the operations of other functions (nor are able to replace a function application by another one). That is, no λ-calculi operations, namely $\{\beta$-reduction, α-conversion, η-conversion$\}$ and higher-order function application, are capable of doing this at all, as λ-expressions are always well formed. The same is valid for any functional model. Thus, let sef_u, sef_v, sef_g: $MC \times P^* \times \mathbb{N} \longrightarrow \mathbb{N}$ be the effectively computable functions which are supposed to correspond to U, V and G, respectively, and their corresponding sequences of steps S_u, S_v and S_g. The three sequences of steps depend on their respective effectively computable functions. Finally, while the applications $U(U[G[X]])$ and $V(V[G[X]])$ do not always produce the same value for all G, the corresponding applications $sef_u(\lambda{-}calculus, S_u, sef_g(\lambda{-}calculus, S_g, x)) = u(g(x))$

and $sef_v(\lambda{-}calculus, S_v, sef_g(\lambda{-}calculus, S_g, x)) = v(g(x))$ always result in the same values for all g, regardless of whether $u(g(x)) = v(g(x))$ or $u(g(x)) \neq v(g(x))$ or not, since sef_u and sef_v are functions. By assumption, the absence of one corresponding function for $V(V[G[X]])$ is a contradiction.

□

Briefly, from the presented alternative view of unexpected effects, while in the computation of the Turing-machine composition $V(V[G[X]])$ the computation of the Turing machine G might replace the inner occurrence of the Turing machine V by another Turing machine V' and, therefore, the outermost occurrence of V might compute $V(V'[G[X]])$ instead, neither the steps S_g nor the function sef_g can replace any function, in particular, neither sef_u nor sef_v.

Theorem 2 *If each Turing machine implies one partial function, then the class of Turing-computable functions is not isomorphic to the class of effectively computable functions.*

This theorem is another way of seeing the theorem 1.

□

Because the models of effectively computable functions provide ways of defining functions that result in any number as we wish, then for all $k \in \mathbb{N}$, for all $(x, y) \in \mathbb{N}^k \times \mathbb{N}$, intuitively, there must exist an effectively computable function which calculates y from x in a few steps. However, as I have shown, Turing machines are not necessarily functions. As already mentioned, unexpected effects introduce a problem with two aspects: logical and semantic. I solve the logical aspect of the problem by proposition 2, and I solve the semantic aspect of the problem by regarding Turing machines that produce different answers under different physical conditions as non-functional machines. Thus V is a Turing machine which does not have any corresponding function. Furthermore, it is easy to see that Turing-computable functions are still linked to Turing machines where unexpected effects are forbidden. Notice that this corollary holds for both intensional and extensional standpoints, as we can also view unexpected effect as an action or effect of replacing one Turing machine by another with different results.

Theorem 3 *Computation is not conceptually function application.*

[Proof] Given that computation may be mobile, e.g. by using mobile agents nowadays, given some insight of the present author in Edinburgh (1999), computation is conceptually a physical process.

On the other hand, by theorem 1, the class of Turing machines is not isomorphic to the class of Turing-computable functions. Following this, programs do not correspond to functions. Therefore, computation is not conceptually function application.

□

Proposition 1 *Let $k \in \mathbb{N}$. For every $k > 1$, there exists a k-level Turing-machine composition if and only if some representation of the Universal Turing machine is not in the composition.*

[Proof] Let U be a Universal Turing machine, $M, N : \mathbb{N} \longrightarrow \mathbb{N}$ be two Turing machines with corresponding Turing-computable functions $m, n : \mathbb{N} \longrightarrow \mathbb{N}$, and $x \in \mathbb{N}$, and $X : \mathbb{N}$ be the representation of x on the tape.

The Universal Turing machine, by lemma 2, guarantees the absence of unexpected effects at all levels of its parameters. I can consider the composition $M[N[X]]$. It follows that U must have *direct* control over the operations of N in such a way that, if N tries to modify the operations in the M representation, U detects this unexpected effect and intervenes, for instance, by moving physically the representation of M or N to another place on the tape, to continue the computation of the composition keeping the isomorphism between Turing machines and computable functions. Therefore, because U must have dynamic knowledge about the computation carried out by N, $U(M[N[X]])$ is not really a function application, and therefore some representation of U is not in the composition.

With respect to the converse, setting $M \neq U \wedge N \neq U$ and $X \neq U$, there exists a Turing-machine composition, e.g. respectively $M(N[X])$ above, from which U is absent.

□

Remark: We can capture an intuitive and precise notion of Turing machine model as follows: Let $\mathcal{M}$ be the set of all Turing machines, T be the set of Turing-computable functions, U be the Universal Turing machine and u be the Universal Turing-computable function. Let $X : \mathbb{N}$ be the null-computation Turing machine that corresponds to the 0-ary function (i.e. without any input) that always results in the same value $x \in \mathbb{N}$. Therefore, $u : \mathcal{P}(T) \longrightarrow \mathbb{N}$ (where $\mathcal{P}$ is the ordered power set of a given ordered set), in such a way that the application $m(n(x))$ is equal to $u(m(n(x)))$ and abstractly represented as $U(\{M, N, X\})$ or, more precisely, as $U(s)$ where s denotes the string that encodes M, N and X, together with the write/read head and blank symbols, with the constraint that s does neither start nor finish with the blank symbol. Furthermore, composition is part of the notion of function, and such a representation does not capture the composition of

Turing machines $U(M[N[X]])$. However, there may be applications as well as compositions involving U where there exist such representations with U, both on the tape and outside the tape. Therefore, there exist two different levels of functional abstraction in the Turing machine model of computation.

In other words, on the one hand, we separate what is encoded on the tape from what is outside the tape, by stating that only what is outside the tape is free from unexpected effects, and hence, can be functions. On the other hand, functions do not manipulate the operations of any function. In this way, there are two different levels of abstraction: at one level, only the Universal Turing machine is function and the encoded Turing machines on the tape form a one-level parameter. At another level, there exists a Turing machine composition on the tape, and the encoded Turing machines correspond to the computable functions because the Universal Turing machine does not correspond to any function in the same space, in the sense that U is capable of managing the tape and guaranteeing absence of unexpected effects. Therefore, there exist two separate levels of function abstraction in the Turing machine model of computation.

In the next proposition, as usual, I do not regard time as a factor in the computation.

Proposition 2 *There exists a Turing machine that can correspond to more than one Turing-computable function.*

[Proof] Let M be some Turing machine and x be its input. Let U be a Universal Turing machine, and V be another Turing machine, which, except for the existence of unexpected effects, produces the same output as U: the only difference is that U calculates $U(U[M[X]])$, and V calculates $V(V[M[X]])$ and sometimes calculates $U(U[M[X]])$, but sometimes not, depending on the physical places where V and M rest on the tape. Because the Turing machine composition $V(V[M[X]])$ can be placed at different places on the tape at different instants and the computations receive different kinds of unexpected effects, the same running Turing machine M can produce different results for the same input x. Each particular result from x corresponds to one Turing-computable function.

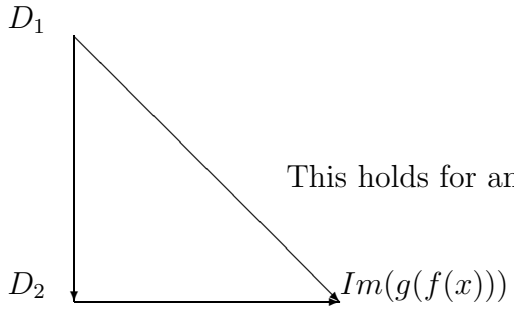
□

Corollary 1 *There exists a Turing machine that can avoid receiving unexpected effects from its parameter.*

[Proof] Universal Turing machines, as discussed in the theorem 1, must ensure absence of unexpected effects.

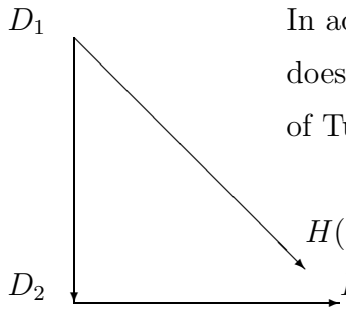
□

I can draw functions $f: D_1 \longrightarrow D_2$ and $g: D_2 \longrightarrow D_3$ as well as the corresponding composition $h: D_1 \longrightarrow D_3$, under the law $h = g(f(x))$ as in the following picture:



This holds for any function application.

(where $Im(g(f(x))) \subseteq D_3$).



In addition to the Tm composition,
does it dynamically hold for any computation
of Turing Machine composition?

Compositions

For answering the question, I individually consider the correspondence between the functions f, g and h, and the Turing machines F, G and H, respectively. The answer for the question, i.e. whether the law of composition $F(G(X)) = H(X)$ holds for every computation of composition of Turing machines, depends on the absolute positions of the involved Turing machines, F and G, on the tape, as well as on whether the only Turing machine outside the tape avoids unexpected effects on the tape. However, both factors are *external* to the machines that are on the tape and play rôles in the composition. As a consequence, the global view is a property of the Universal Turing machines, in particular, it avoids what I discovered in 2000 and refer to as unexpected effects.

In this article, as well as in [4], the term *algorithm* is used to mean a program which always halts, and not necessarily a total function. A decision had to be made.

References

[1] G. S. Boolos and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, third edition, 1989.

[2] N. Cutland. *Computability: an introduction to recursive function theory*. Cambridge University Press, 1980. This book was reprinted.

[3] U. Ferreira. On Turing's proof of the undecidability of the halting problem. In H. R. Arabnia, I. A. Ajwa, and G. A. Gravvanis, editors, *Post-Conference Proceedings of the 2004 International Conference on Algorithmic Mathematics & Computer Science*, pages 519–522. CSREA Press, June 2004. Las Vegas, Nevada, USA.

[4] U. Ferreira. A property for Church-Turing thesis. In H. R. Arabnia, I. A. Ajwa, and G. A. Gravvanis, editors, *Post-Conference Proceedings of the 2004 International Conference on Algorithmic Mathematics & Computer Science*, pages 507–513. CSREA Press, June 2004. Las Vegas, Nevada, USA.

[5] A. Galton. *Machines and Thought: The Legacy of Alan Turing*, volume 1 of *Mind Association ocasional series*, chapter The Church-Turing Thesis: Its Nature and Status, pages 137–164. Oxford University Press, 1996.

[6] N. D. Jones. *Computability Theory: An Introduction*. ACM Monograph Series. Academic Press, New York and London, 1973.

[7] N. D. Jones. *Computability and Complexity: from a programming perspective*. Foundations of Computing. The MIT Press, 1997.

[8] S. M. Lane. *Categories for the Working Mathematician*. Graduate texts in mathematics. Springer, second edition, 1998. Previous edition: 1971.

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

[10] M. Margenstern. On quasi-unilateral Universal Turing machines. *Theoretical Computer Science*, 257(1–2):153–166, April 2001.

[11] A. Mateescu and A. Salomaa. *Handbook of Formal Languages*, volume 1, chapter Aspects of Classical Language Theory, pages 175–251. Springer-Verlag, 1997.

[12] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Series in Automatic Computation. Prentice-Hall International, Inc. London, 1972. Original American publication by Prentice-Hall Inc. 1967.

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

[14] I. C. C. Phillips. *Handbook of Logic in Computer Science*, volume 1: Mathematical Structures, chapter Recursion Theory, pages 79–187. Oxford University Press, 1992.

[15] B. C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. The MIT Press, 1993. Second print.

[16] A. Poigné. *Handbook of Logic in Computer Science*, volume 1: Mathematical Structures, chapter Basic Category Theory, pages 413–640. Oxford University Press, 1992.

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

[18] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley Publishing Company, second edition, 1999. Paperback.

[19] J. V. Tucker and J. I. Zucker. *Handbook of Logic in Computer Science*, volume 5: Logic and Algebraic Methods, chapter Computable Functions and Semicomputable Sets on Many-Sorted Algebras, pages 317–523. Oxford University Press, 2000.

[20] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1936.

[21] A. Yasuhara. *Recursive Function and Logic*. Academic Press, Inc, 1971.