A Property for Church-Turing Thesis

by Ulisses Ferreira

***Abstract** - This paper proves that, unless a Universal Turing machine is involved in compositions with some particular property, the class of Turing-computable (partial) functions is not isomorphic to the class of effectively-computable (partial) functions.*

1 Introduction

Since 1930's, there has been interesting work into foundations of computer science, in theory of computation [8], category theory[7, 14] as well as in recursive function[1] theory, functional programming[17] and other theoretical subjects. To date, nobody seems to have observed unexpected effects in computations of compositions of Turing machines on the tape. 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 replacing F by 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 article, we shall see such details in a careful and more precise way. Variations on Universal Turing machines have been proposed[9] but not much work has been carried out in the only and traditional computability theory.

In the next section I present Turing machines from the operational standpoint.

2 Turing machines

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. 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:

[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 functions in the present article are total or partial, 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.

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 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 known actions.

Just a short note on notation used in the present article, 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.

Here, we can define the alphabet $\Sigma_2 \overset{def}{=} \Sigma \cup \{\odot\}$ 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 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.

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[2] 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.

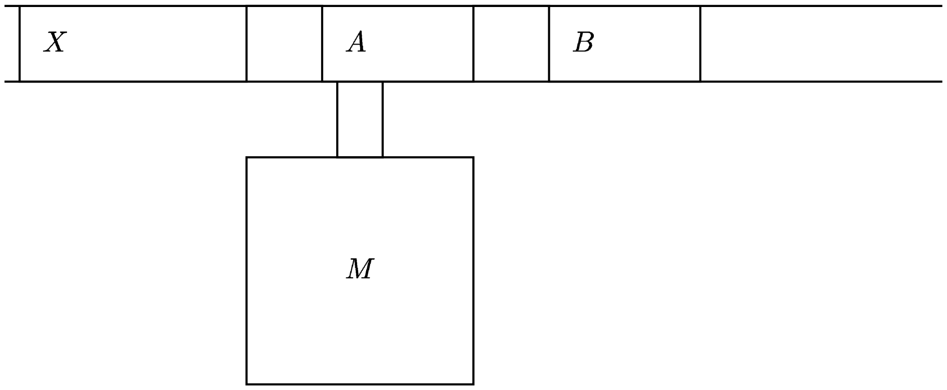
Below, I define the Universal Turing machine with the characteristics that will be subsequently helpful in this article.

Definition 1 (Universal Turing machine) *Let U be a Turing machine. 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

[2]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.

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:



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 encoding 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.

2.1 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.

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.

3 A Refuting Example

In the following theorems of this article, 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, where $\diamond$ indicates the write/read head:

tape starts here $\longrightarrow|\underset{\diamond}{1}011101 \circ ... r(G) \circ ... r(F) \circ ...$

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. $\square$

Definition 2 *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. $\square$

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 garantees 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 programable 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. $\square$

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 wrt 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 $\longrightarrow \underset{\diamond}{|}1011101 \circ ...r(G) \circ ...r(U) \circ ...$

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

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

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. $\square$

In other words, in the presence of compositions, the model of Turing machines is not equivalent to the model of effectively computable functions.

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] Ulisses Ferreira. On Turing's Proof of the Undecidability of the Halting Problem. In Hamid R. Arabnia, editor, *Proceedings of 2004 International Conference on Algorithmic Mathematics and Computer Science*, June 2004.

[4] 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.

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

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

[18] 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.

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

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