The Probable Decidability of the Halting Problem

Ulisses Ferreira
http://www.ufba.br

Abstract ***In the present paper, I present an epilogue of the past works on the halting problem, also known as a kind of the Entscheidungsproblem over Turing machines, although this paper is relatively short. As part of this work, I continue some work by the present author on the busy beaver problem, because the busy beaver problem was introduced (in 1962) over the former, thus endorsing the belief of the undecidable halting problem, and there has also been a general and strong belief that the latter is also undecidable. They are not dependent from each other though. Finally, I present two different proofs.***

Keywords: The halting problem

1 Introduction

Paper [4] reports that some fundamental mistake had been discovered. The next step is to attempt to prove the decidability of the busy beaver problem, as well as, consistently, to attempt to prove the decidability of the halting problem, and those are what I do in the next sections.

2 The possible decidability of busy beaver problem

For each computable function application here, the corresponding Turing machine places the write-read head on the leftmost non-blank symbol, thus leaving the situation on this *standard configuration*. If the machine does not do so, here the computation is regarded as partial. The same holds for every computation that does not halt. The interpretation is that the computable function does not have that input value in its domain.

The proposed Turing machine either sequentially writes a number of ones or multiplies the current number of ones by some positive natural number. Then, as the alphabet is binary, the number of states are added to this to form the resulting value. Moreover, I have to take into account that the Turing machine may write sequentially either in those first applications, or during them, for I am looking for the greatest value. Let Λ be the computable function that corresponds to the longest string of ones written by the Turing machine on the tape, let $\zeta \in \mathbb{N}$ be the minimum number of states for duplicating any string given one Turing machine definition (in this way, $(\zeta + 1)$ is the minimum number of states for multiplying the length of the string by 3, and so forth. More precisely, $\zeta + k$ can be interpreted as some arrangement of states and tuples where, in any such computation, for each iteration, the corresponding machine writes k ones on the tape for multiplying the input by the $(k + 2)$ factor, although some other arrangements with the same number of states may yield better results and overwrite this one[1]), and the function $B : \mathbb{N} \longrightarrow \mathbb{N}$, as an approximation to Λ, can be finally defined as follows:

$$B(t) = \begin{cases} 0 & \text{if } t = 0; \\ \Pi(t, 0) & \text{if } t > 0. \end{cases} \tag{1}$$

for number of states t and which applies the following function

$$\Pi(a, b) = \begin{cases} \lambda(a, b, 0) & \text{if } a = 1; \\ m(\lambda(a, b, 0), \Pi(a - 1, b + 1)) & \text{if } a > 1. \end{cases} \tag{2}$$

which in its turn applies the following maximum function

$$m(a, b) = \begin{cases} a & \text{if } a \geq b; \\ b & \text{if } a < b. \end{cases} \tag{3}$$

as well as the following one

[1] Notice that, here, because Turing machines are not equipped with the notion of subroutine, for each function application, the machine consumes $\zeta + k$ states. Therefore, I have to consider that states are disposable, even for the same function definition.

$$\lambda(a,b,c) = \begin{cases} a+b & \text{if } b < \zeta + c; \\ a+b & \text{if } b = \zeta + c \wedge \\ & a+b \geq a(c+2); \\ a(c+2) & \text{if } b = \zeta + c \wedge \\ & a+b < a(c+2); \\ m(\lambda(& \\ a+\zeta+c, & \\ b-\zeta-c, & \\ c), & \\ \lambda(a,b,c+1)) & \\ & \text{if } b > \zeta + c \wedge \\ & a+\zeta+c \geq ac+2a; \\ m(\lambda(& \\ a(c+2), & \\ b-\zeta-c, c), & \\ \lambda(a,b,c+1)) & \\ & \text{if } b > \zeta + c \wedge \\ & a+\zeta+c < ac+2a \end{cases} \tag{4}$$

where a denotes the number of states for writing the string sequentially, b denotes the number of states available for applying multiplications on the string that rests on the tape, c is a multiplicative operand, and t, as already stated, is the total number of states available for one Turing machine.

Clearly, B is a computable total function, and the present author regards that the solution of the busy beaver problem is certainly in $[B(x), B(x)+x]$ interval, and that $B(x)+x$ as solution is rare if possible, in particular if the machine is defined to compute leaving standard configurations. Clearly, the solution is in $[B(x), B(x+1)]$, probably very close to $B(x)$ as x grows. In this way, letting $BB(x) = B(x) + x$ be this asymptotic function that is not lesser than the solution of the busy beaver problem, for the Turing machine defined in [1], $\zeta = 11$ in accordance with [1].

Note that the above function requires that the Turing machine, for the solution of the busy beaver problem, leaves on the tape only one sequence of ones. From the definition in [1], the scores will be different from the present ones maybe due to their standard configuration, whereas, in [3], the scores were greater than the present ones for there was blank symbol occurrences in result. For this case, one might use the Λ function defined in section 5.

3 A novel adaptation: A Finite-Tape Model

In this section, I describe a finite-tape model which will be called here $(n)FTM$. It was initially conceived in 1999 and written in the year 2000, and the only relevant difference from the model of Turing machines was the finiteness of the tape. In 2002, I added a new feature to that which is an interactive operation.

A Finite Tape Turing Machine with n squares, referred to as $(n)FTM$, is just a Turing machine supplied with n squares on the tape, and, for variations of the present finite model, of mine, there is some interpretation for the attempt to move the write-read head outside the tape, e.g. that the machine does not halt. One of the best formal definitions of Turing machines is in [6]. Here, I make a brief modification.

Definition 1 A Turing machine with tape containing n squares, $(n)FTM$, is an ordered system $M = (Q, \Gamma, \Sigma, \delta, q_0, \sqcup, F, n)$ where Q is a finite set of states; Γ is the input alphabet; Σ is the tape alphabet; constraints $\Sigma \cap Q = \emptyset$, Σ is finite and $\Gamma \subset \Sigma$; $q_0 \in Q$ is the initial state; $\sqcup \in \Sigma - \Gamma$ is the blank symbol, $s_0 = \sqcup$; $F \subseteq Q$ is the set of final states, therefore also finite; and δ is the transition function

$$\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{\leftarrow, \rightarrow, \triangle, \Omega\} \tag{5}$$

□

In accordance with what is suggested, there are two characteristics to be added to the Turing machine, namely, finiteness of memory, as well as interactive operations, both input and output. Whereas the former modification is timeless and represents the limitations of physical resources of a machine, the latter is temporal and has a motivation to represent interactive computation, in particular, after that ancient notion of data processing in batch. The above definition partially adds the latter characteristics to the model while I use a tape with n squares.

With respect to the input/output operations, intuitively, I can solve this problem in the following way: add to a Turing machine a unique operation for input and for output Ω to the set of operations $\{\leftarrow, \rightarrow, \triangle\}$[2], as shown above, in a way that after having recognized Ω in the image of δ, the

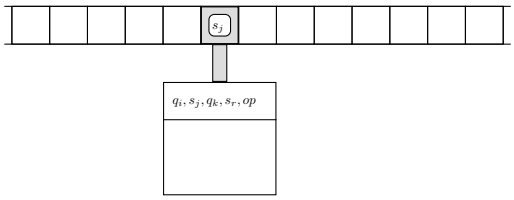
[2] $\{\leftarrow, \rightarrow, \triangle\}$ represents, respectively, the operations shift the write-read head one square to the left, shift the write-read head one square to the right, and finally make the write-read head stay put.

machine temporarily halts, shows the symbol under the write-read head in some output device, and wait for an input symbol in Σ (that is, an external action, say from a user), as well as the machine replaces the previous symbol by the one from the external action without moving the write-read head. This operation does not change state either, the value of the new state in the image of δ is simply ignored in this case. Therefore, the operation Ω changes only the current symbol on the tape. In this way, the user can type the new symbol shown in this output device when the Turing machine is only displaying a value (that is, the programmer presumes that the user does not wish to change it), or type a different symbol, in case of input operation (that is, the programmer presumes that the user wishes to change it). Although this kind of unified operation is not usual in the real world, it simplifies the model without loss of generality.

If the interaction operation is in a state in F, then the Turing machine computation halts without performing the interaction.

Initially, the write-read head is on the left-most square of the tape, also on the left-most symbol. I refer to this square as *square 1*. An input string is said to be accepted by the Turing machine if and only if, after a finite number of steps, the machine $(n)FTM$ reaches one of its final states, in F. In case there is no transition for the current state and symbol currently under the write-read head, formally, $(\forall q \in Q)(\forall s \in \Sigma)\ (q, s) \notin dom(\delta)$, the machine performs infinite computation and hence no longer gives any response whatsoever. In this case, the result from the computable function is said to be undefined, with usual notation $\perp$ if one wishes to refer to it.

In case of an operation in $\{\leftarrow, \rightarrow\}$, I can establish here that, in case the computation tries to move the write-read head outside the range of the tape, the machine simply does not move the write-read head in that step and the computation continues, but the $(n)FTM$ can be different. As a simple matter of choice, here, computation is neither sound not complete with respect to the original Turing machine. If, instead of only refusing to move the write-read head the machine refused to continue computing, the computation would be sound but not complete with respect to Turing machines as originally defined. In the remaining cases, moving the write-read head to the left corresponds to decreasing that current numeric square position. Accordingly, moving the write-read head to the right corresponds to increasing that current numeric square position. I could think of running $(n)FTM$ in state q_i and with the square under the write-read head containing the symbol s_j as in the following drawing:



A Finite Turing Machine with 13 squares being shown

In a way similar to a conventional Turing machine C, for the effect of the $(n)FTM$ machine M not moving the write-read head outside the tape boundaries, I consider two squares on the C tape, namely square 0 and square $n + 1$, together with the n squares, from 1 up to n. A particular symbol of an alphabet is said to be a *reserved symbol* if it is used for one purpose and, in order to make the idea be successful, that symbol cannot be used in other contexts. In this way, the left-most square becomes the square of number 0 and it always contains a reserved symbol $\triangleright$ which is not in the alphabet of M, and the right-most square becomes the square of number $n + 1$, and that square contains the reserved symbol $\triangleleft$ which is not in the alphabet of M either. Once the write-read head has reached the square 0 or the square $n + 1$, to make the machine no longer gives the answer, I just obtain $(\forall q \in Q)\ \delta(q, \triangleleft) = \delta(q, \triangleright) = \perp$, or in other words, there is no transition for either symbol. However, in this case, there exists one or more transitions that move the write-read head, currently on the $\triangleright$ symbol, one square to the right, or *infinitely* to the right, in accordance with how I define a $(n)FTM$; Accordingly, there is one transition that moves the write-read head, currently on the $\triangleleft$ symbol, one square to the left or, alternatively, two transitions that move infinitely to the *right* according to the definition, and I chose to have one transition that moves the write-read head one square to the left. For students, the number of transitions may be approximated and depends upon the particular $(n)FTM$.

Alternatively, for preventing the effect of moving the write-read head outside an imaginary finite tape on Turing machines, I do the following: for every state q_i, I construct a tuple $(q_i, \triangleleft, q_i, \triangleleft, \leftarrow)$ of Turing machine C, and the symmetry applies for the $\triangleright$ symbol.

From now on, I shall abstract the above details of interpretation. The left-most square is again the number 1 and the right-most square is the n. The above interpretation has the purpose to show that a $(n)FTM$ and a Turing machine are approximately the same model, but those differences are theoretically interesting.

I observe that a $(n)FTM$ implies that data represented by the content of the tape is always a string with length n, and the blank symbol may occur in the string. The number of complete expressions on such a machine is the following:

$$n \times |Q| \times |\Sigma|^n \tag{6}$$

therefore, a finite number of possible situations. Initially, as already set, the write-read head is on square 1.

Definition 2 *A* complete expression *is a record composed of the following variables:*

- The sequence of symbols of the Turing machine which are on the tape, in some finite representation;
- The current state q;
- The write-read head position on the tape.

□

In any simulation, the complete expression repeats if and only if the simulated computation never halts.

In this section, I showed by giving examples how a $(n)FTM$ can be a finite model of serial computation based on Turing's model. In the following section, I shall consider interaction for $(n)FTM$.

4 Two points of view, the two halting problems

There are two points of view on the halting problem, as stated in the following subsections.

This section makes use of an intuitive term, namely *computational limit*, which will be informally defined from here on.

4.1 The real halting problem

It is known that real and physical machines also have their own computational limit, although they are not in terms of tuples. Therefore, the referred to P program running on a real machine cannot decide over any computation that requires some machine with larger computational limit.

4.2 An equivalent point of view

That looks as if M were outside the tape and only X were written on the tape while $\mathcal{H}$ were only in a human mind and outside the represented objects. This point of view is equivalent to the busy beaver problem, since the latter consists in finding a function only. The halting problem normally assumes that $\mathcal{H}$ is outside the tape, and both M and X at some established positions are already printed on the tape.

5 My Proof

In this section, I prove the decidability of the second version of the halting problem.

Definition 3 Let $M = (Q, \Gamma, \Sigma, \delta, q_0, \sqcup, F)$ be any arbitrary Turing machine, let X be its input data placed on the tape and $\sqcup \notin \Gamma$. It is referred to as *computational limit* of $M[X]$ the maximum number of different squares that can be read or written by $\uparrow M[X]$, apart from X, such that $\uparrow M[X]$ halts.

□

As an example of Turing machine computational limit, if the tape contains only the blank symbol, if a Turing machine contains only one state, q_0, as well as the final one, q_f, no matter the cardinality of the alphabet, it cannot write on more than a single square before moving to q_f. Therefore, the computational limit is 1 for every such a Turing machine and data. The same limit holds for, if instead, the tape initially contains one non-blank symbol. This happens since the notion of Turing machine computational limit does not include those squares where X initially rests.

Given any alphabet Σ and any set Q of states, the Turing machine M can be built for visiting the squares of the tape as many as possible and finally halting. Without loss of generality, let Σ be a partially ordered set where $\sqcup$ is the maximum element of the partial order, or formally $\forall x.\ x \sqsubseteq \sqcup$, and let us assume that, as well as both infinite sequences of $\sqcup$, one for each side on the tape, the tape contains finite occurrences of the least symbol only, that is s such that $\forall x \in X.\ s \sqsubseteq x$ [3].

Exponential, in terms of pure functions, is the greatest function and any machine or person cannot produce a value greater than the above, since the parameters correspond to resources. A greater value requires greater values of parameters. A

[3] Please, notice that I am looking for the maximum value for a previously defined Turing machine.

strong piece of evidence towards the established idea of Turing machine computational limit is the known results[5] for the busy beaver problem, for instance, $BB(1) = 1$, $BB(2) = 4$, $BB(3) = 6$ and $BB(4) = 13$, which imply a monotone increasing function. Now, there is no reasons why the theoretical community should not believe that there are computational limits for Turing machines with their corresponding input.

The computable function of t that corresponds to the computational limit in number of squares is now defined as follows:

$$\Lambda(\Sigma, X, t, n) = 2 \times |\Sigma|^{|X|+t+n+2} \tag{7}$$

where X is the input data string, t is the number of tuples and n is the number of states of any Turing machine. However, in terms of number of steps, the referred to computational limit gets the following:

$$\Phi(\Sigma, X, t, n) = 2 \times n \times |\Sigma|^{2\times(\ |X|+t+n+2\ )} \tag{8}$$

and one may unify these two notions by using only the larger one instead of two variables or functions, for the present work is not concerned for efficiency issues. Stating that there is no function that returns greater values than Λ and Φ above, means that, given any arbitrary computable function, say with binary notation $\times$ in the form of an operator[4], the only manners to define a value greater than the one from any exponential function, say generically x^y, which is the essence of Λ or Φ, is by defining another operand or function, let us use f function abstractly to be such an operand, to form a combination of the following expressions: $f \times x^y$, or $x^y \times f$, or $(f \times x)^y$, or $(x \times f)^y$, or $x^{(f\times y)}$ or $x^{(y\times f)}$. However, in any case, this hypothetical greater value requires extra number of states and tuples. Moreover, it is clear that the $\times$ function cannot be included to the referred to exponential function because there is not more number of states available. Thus, I am referring to a set of tuples with the number n previously defined as a parameter.

The idea of one maximum value is the same as the used for the busy beaver problem.

Regarding the minimum number n of tuples for writing the Turing machine that can compute Λ and Φ for one side of the tape given Q and Σ, the present author uses $7 + |\Sigma|$ tuples for n in the base case as follows, setting the starting symbol as A, $\Sigma = \{0, ..., 9, \sqcup\}$ and $F = \{Z\}$:

$$\{(A,'0', B,'1', \leftarrow), (A,'1', B,'2', \leftarrow), ...,$$
$$(A,'8', B,'9', \leftarrow), (A,'9', A,'0', \rightarrow)\}$$

[4]Here, $\times$ is not necessarily the multiplication operator.

$$\cup \tag{9}$$
$$\{(A, \sqcup, Z,'0', \triangle), (B,'0', B,'0', \leftarrow),$$
$$(B, \sqcup, C, \sqcup, \leftarrow),$$
$$(C,'0', C,'0', \leftarrow), (C, \sqcup, D,'0', \rightarrow),$$
$$(D,'0', D,'0', \rightarrow), (D, \sqcup, A, \sqcup, \rightarrow)\}$$

Thus, with $\kappa(7 + |\Sigma|)$ tuples, for any $\kappa \in \mathbb{N},\ \kappa > 0$, there may be the referred to recursive case with adaptation in the set of tuples replacing the transition to the final state by a transition to the state that is supposed to correspond to the initial state of the next subset of tuples, while the last subset inherits the corresponding transition towards the final state. It is clear that the implementation of recursion in a Turing machine does not consume any extra state.

6 Does Uncomputable Function Exist?

In this section, I observe that uncomputable functions do not exist. Although this section was really unrefereed, it is certainly worth presenting it here, as this particular result is another way to refute the idea of the undecidable halting problem.

It is known[1, 2, 7] that, unary computable functions can form a denumerable set according to some enumeration of Turing machines. Therefore, let such an enumeration of all of the computable functions be with the following monadic symbols f_i for $i \in \mathbb{N}$: $\{f_1(x), f_2(x), f_3(x), ...\}$, from natural numbers to natural numbers. Let us define $g : \mathbb{N} \longrightarrow \mathbb{N}$ as follows:

$$g(x) = \begin{cases} 0 & \text{if } f_x(x) \text{ is undefined;} \\ f_x(x) + 1 & \text{if } f_x(x) \text{ is defined.} \end{cases} \tag{10}$$

Clearly, $g(x)$, above, has been regarded as a total function. Thus, if one assumes that all total unary functions are computable, one obtains a contradiction: On the one hand, for every $n \in \mathbb{N}$, $g(n)$ seems to be different from $f_n(n)$ when $f_n(n)$ is undefined, because g is defined, above. On the other hand, $g(n)$ differs from $f_n(n)$ when $f_n(n)$ is defined because $g(n) = f_n(n) + 1$ and the results are also different for all n. And if g differs from f in all cases, g does not belong to the enumeration. However, as one assumed that all total unary functions were computable, one also assumed that g would be in the enumeration, and so one arrived at a contradiction. Therefore, the conclusion is that there exists some total unary function that is not computable, and that the conclusion is correct.

6.1 However, Another View

I see the picture in a different way.

If the halting problem is unsolvable as it has thought to be, despite the appearance, the above $g(x)$ function is *not* a total function but instead a *partially computable function* since its value depends on the value of $f_x(x)$ which in turn has to be calculated beforehand, and can be undefined. In this case with $x = n$, the partially computable $g(x)$ corresponds to a Turing machine which loops in the case of the particular value n.

On the other hand, if $f_x(x)$ is defined, the partial $g(x)$ is also defined, and $g(x) = f_x(x) + 1$, like in the other view, and there is no contradiction while $g(x)$, as partial, is not in the referred to list.

Thus, *there would be no uncomputable function* as shown below, even for the halting problem unsolvable.

6.2 However, If the Halting Problem is Decidable?

Let us check how the above computation of $g(x)$ may be thought to be performed. Initially, the code of g is *outside* the tape (this Turing machine will be called G) while the tape contains only a particular value of x in some representation X. As soon as the machine G starts running, it transforms the particular value of x into some $F_x(X)$ (some representation of $f_x(x)$), obtaining the representation of x and of $f_x(x)$, both on the tape. Then, G starts interpreting $F_x(X)$ until one obtains its result if any, with the value of the function application $f_x(x)$. Finally, if the computation of $F_x(X)$ halts, the result is added to 1 as the result of $G(X)$. Otherwise, the result of $G(X)$ is 0. Here, I assume the absence of any unexpected effect during that interpretation.

A second scheme is the following: Initially, the Universal Turing Machine U is outside the tape, while both X and $G(X)$ are encoded on the tape. For a Universal Turing machine that always guarantees the absence of side effects, the result is always exactly the same as in the above paragraph.

There is still something wrong in the above computation. If we checked it in a deeper way, we would observe that, in the case that $g(x) = f_x(x)$, the computation with all its details would run forever during the decoding process. In other words, when $F_x(X)$ is interpreted, a new copy of $F_x(X)$ is created, then this new copy is started being interpreted, another copy appears and so on. Therefore, in the particular case of $g(x) = f_x(x)$, no matter other parts of the definition of g, the corresponding computation unconditionally would not halt and, therefore, g is a partial recursive function, not an uncomputable function.

Another way of seeing the same picture is that no Turing machine can decode itself and then start interpreting itself to obtain any result, as there will be no result.

Assuming the Church-Turing thesis, since G never halts for $g(x) = f_x(x)$, G actually corresponds to a computable function, or the Church-Turing thesis is not valid.

Note that, as a consequence of this assumption, the self halting problem is solvable.

In the above known proof, that deduction does not prove that there is uncomputable functions, but instead that no functions can unconditionally encode or decode itself as part of its computation with a result.

If the halting problem is published as solvable as the present work shows, for all x, $g(x)$ is defined where $f_x(x)$ is undefined and, in this case, results in 0. Moreover, if $f_x(x)$ is defined, the computation of the g function becomes equivalent to $g(x) = f_x(x)+1$.

Note that, in the first case of the $g(x)$ function[5], where $f_x(x)$ is undefined, $g(x)$ is defined if and only if one algorithm that solves the halting problem is properly involved in the computation of $g(x)$. More generally, in the same view, any function $g(x)$ is regarded as total if and only if some algorithm deciding the halting problem is in the computation of $g(x)$ wherever another function is undefined.

Here, regardless of whether the halting problem is decidable, the function $g(x)$ is in the above list of computable functions. However, in the particular case where $g(x) = f_x(x)$, g is undefined for its computation never halts.

If I consider the present refutation of the proof of the unsolvability of the halting problem as above, the extent to which the Cantor diagonal process can be certainly applied to the computability theory, whether the process should really be applied whatsoever, is a question that needs to be thought, at least for this claim. However,

Any natural function is computable:

Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be any function for any $k \in \mathbb{N}$ such that $k \geq 1$. Therefore, by definition, f has one domain $Dom(f)$, one codomain $Cod(f)$ and one image $Im(f)$, where $Im(f) \subseteq Cod(f)$. No matter how f is *represented* and whether f *is defined* as recursive or not, f is a function and can be described as a set of tuples where each tuple has $k + 1$ elements

[5] See the definition of the formula 10.

in $\mathbb{N}$ where the first k elements of every tuple (in some previously set order) denote one element of $Dom(f)$ while the $(k+1)$th element is the result from f and denotes one element of $Im(f)$. In this way, the referred to sequence shows only those elements where f is defined. Where f is undefined, the tuple is not in the sequence.

Any set of tuples such that each tuple has k natural elements is a countable set. Clearly, some computation of f given one element in $Dom(f)$ is trivial searching the sequence in this countable order.

Acknowledgments

Many thanks to P. Chaves.

7 Conclusions

The halting problem is decidable.

References

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

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

[3] A. K. Dewdney. A computer trap for busy beaver, the hardest working Turing machine. *Scientific American*, 251:16–23, August 1984.

[4] Ulisses Ferreira. On Turing's proof of the undecidability of the halting problem. In Hamid R. Arabnia, Iyad A. Ajwa, and George 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.

[5] Jozef Gruska. *Foundations of Computing*, chapter 4.1.6 Undecidable and Unsolvable Problems, pages 227–229. International Thomson Computer Press, 1997.

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

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