JOURNAL OF COMPUTER AND SYSTEM SCIENCES **45**, 367–384 (1992)

Foreground Automata*

IAN F. CARLSTROM

Department of Philosophy, John Carroll University,
University Heights, Cleveland, Ohio 44118

Received April 16, 1990; revised September 2, 1990

This paper defines a class of on-line *foreground automata*, which make distinctions between the "foreground" or relevant inputs and outputs and the "blank" ones that serve as a background. It is shown that there is a well-defined operation that maps the substring of relevant inputs into an eventually appearing substring of relevant outputs, without regard for the blanks scattered among the inputs. This operation plays the role of the computation of an off-line automaton and a computational time can be measured by comparing the automaton to a "benchmark automaton" that produces each relevant output as soon as theoretically possible. Properties of these computational times are explored, both for finite automata and "Turing automata," which are modeled by multi-tape Turing machines. An analogue of Church's Thesis can be stated for the computations associated with the operations of Turing automata, but it is argued that there is no clear cut formalization for the concept of an "effective foreground automata." © 1992 Academic Press, Inc.

1. INTRODUCTORY REMARKS

A deterministic physical device which interacts with the world is always receiving some sort of input and providing some sort of output. It would be natural to model it as an "on-line" automaton that receives an input and provides an output at each unit time. Nevertheless, such a model poses a technical and conceptual problem. The requirement that a new input be received and a new output be generated at each time appears to provide the automaton with no time to compute an output and thus no way to have a temporal component for the computational complexity of its response. This paper proposes one possible solution for this apparent dilemma. For an appropriately defined class of "foreground automata" that make distinctions between "foreground" or relevant inputs and outputs and irrelevant or "blank" inputs and outputs which serve as a background, these background "blanks" can fulfill our need to describe the device as always having some sort of input–output interaction while still providing the automaton with the computational time to respond to the relevant data that appears in the foreground of the

* This research was supported in part by a research grant from the Office of Naval Research. The author wishes to thank R. J. Nelson and J. A. Mann for helpful discussions and comments on an earlier version of this paper.

data stream. Under the conceptual scheme that emerges from such a distinction, particular classes of automata, such as finite automata, are capable of performing more computationally complex tasks than otherwise would be possible. Indeed, the very concepts of computational tasks, computational time, and computational complexity take on a new perspective. A computation is now viewed as an operation which maps the relevant portion of the input stream into the relevant portion of the output. Computational time is no longer directly measured as the time used to carry out a computational process. Instead, it arises from certain input and output blanks that serve to retard the eventual appearance of the relevant outputs.

This theory could be applied to a situation where first inputs are received, then a calculation is performed, and finally outputs are produced. However, it becomes far more interesting and useful when blank and relevant inputs are thoroughly intermingled and we need a more subtle analysis of the distinct ways that the automaton responds to these two types of inputs. As an informal motivation, imagine someone attempting to carry on a conversation in a newly acquired foreign language. Such conversations will be far more successful when the other person speaks quite slowly. Here it is easy to make a distinction between the foreground speech that appears among the background pauses in the words and sentences. It would be natural to suggest that the slower speech provides the individual with time required to perform the conceptual processing of the new language.

This paper proposes a formal implementation of these ideas. It defines a class of *foreground automata* that treat blanks in a way that allows us to define a *foreground operation* which maps the substring of relevant inputs from a *proper* string into the substring of eventually produced relevant outputs. Here a "proper" input string for a given automaton must present the relevant data at a slow enough rate for the automaton to react properly. This operation on the relevant data will now be taken as the "computational behavior" in place of the mere input–output mappings. Again the foreign language example provides a natural informal motivation. It does not matter how slowly the other person speaks as long as it is sufficiently slow as to allow understanding. A long paragraph delivered at high speed would only leave the poor listener dazed and confused.

Taking the computational behavior of the automaton as restricted to its operation on the relevant "foreground" data requires a new analysis of computational complexity and *computational time*, in particular. One automaton takes less computational time than another operationally equivalent one when the relevant output is produced more quickly with fewer blanks. It is shown that a foreground automaton is always operationally equivalent to some "maximally efficient" automaton that can operate without taking computational time and can thus serve as a benchmark for measuring such time. Computational time arises from delays caused by blanks in the input and output strings. Thus it can be decomposed into *input delays* and *output delays*. It will be shown that although finite foreground automata cannot always avoid taking computational time, they are able to avoid output delays.

Turing machines with one-way input and output tapes are capable of modeling the "on-line" automata considered in this paper and will be called "*Turing automata.*" In effect, a foreground Turing automaton operates as what Arbib [1, Chap. 1] calls an "off-line" Turing machine, while still acting as an "on-line" automaton. Church's Thesis, which states that every effective function can be computed by a Turing machine, can be extended in a purely natural way to the thesis that every effective function can be computed by the foreground operation of a foreground Turing automaton. This further illustrates the natural way in which these string operations capture our informal concept of a "computation." Well-known results from the theory of computational complexity of the Turing machines, such as the speedup theorem [2, Theorem 12.14], can be used to show that Turing automata cannot always operate without using computational time, so that the corresponding benchmark automaton cannot be a Turing automaton. When we seek some analogue of Church's Thesis for the concept of an "effective foreground automaton," it becomes clear that there is no readily defined class of automata that would serve the purpose. It is conjectured here that there is no such analogous formalization for this concept.

The theory will be cast in terms of the external input–output behavior of automata without reference to specific internal states. It might seem natural to utilize what Arbib [1, Chap. 1] calls "length-preserving sequential functions," which are concatenation preserving and map a finite string from the input vocabulary to a string of the same length from the output vocabulary. However, this familiar characterization of automata behavior is inadequate for our present purpose. We are interested in the "operational behavior" of foreground automata, i.e., the mapping from the substring of relevant inputs into a substring of eventually produced relevant outputs. Upon receiving a finite string of relevant inputs, the automaton will eventually produce the determined string of relevant outputs, but often only at a later time, after it has received additional relevant inputs. An attempt to cast such behavior in terms of finite string functions would be quite awkward, and it proves to be convenient to use infinite strings instead. We will use the term "string" to refer to such infinite sequences. The behavior of an automaton can be characterized by its *string function*, the mapping that it induces from infinite input strings to infinite output strings. A reader familiar with the concept of a sequential function should note that the two concepts are easily interdefined.

The paper is organized as follows. Section 2 introduces the basic definitions of a *foreground automaton*, a *proper input string*, and a *foreground string operation*. Here it is shown that there is a well-defined foreground string operation for a foreground automaton with a proper input string. Section 3 introduces a measure of computational time in the performance of a foreground string operation by a foreground automaton. It is shown that there is a "maximally efficient" automaton that can be used as a benchmark for such a measurement. This computational time is decomposed into *input delays* and *output delays* that arise from certain blanks that appear in the input and output strings. The computational times of finite automata are explored in Section 4. Here it is shown that finite automata can always avoid output delay, but that input delay is sometimes unavoidable. The final section introduces a class of *Turing automata*, which are Turing machines that can model foreground automata. The concept of an effective foreground operation is shown to be well defined. Such an operation can be carried out by a Turing automata with just one computational tape. However, doubt is shed on whether there is any analogue of Church's Thesis for a concept of an "effective foreground automaton."

2. The Formal Theory of Foreground Automata

Definition 2.1. (1) An *automaton* M is a quintuple $\langle X, Y, Q, s, r\rangle$, where:

X is the finite input alphabet set,

Y is the finite output alphabet set,

Q is the set of internal states with q_0 in Q the initial state,

s: $Q \times X \to Q$ is the *next-state function*,

r:$Q \times X \to Y$ is the *current-output function.*

Both X and Y are assumed to contain a special symbol "b," called a "blank."

(2) The *computational behavior* of the automaton is defined through a finite sequence of times, beginning with time 0 when the machine is in state q_0. If the machine is in state q at time p and the input at time p is x, then the output at p is $r(q, x)$ and at time $p+1$ the machine enters state $s(q, x)$. Thus the automaton acts on any finite input sequence from X to produce an output sequence from Y of the same length.

We will use the Greek letter ω to denote an infinite sequence ordered as the natural numbers. Those ω-sequences from the set X will be called "input-strings" and ω-sequences from the set Y will be called "output strings." We will use the notations $[\alpha]_p$, for the *initial segment* of string α of length p, and $(\alpha)_p$ for the *pth member* of the string, called the "value of α at time p."

Definition 2.2. A function F that maps input strings to output strings is a *string function* when it satisfies the condition that whenever $F(\alpha) = \beta$, $F(\alpha') = \beta'$, and $[\alpha]_p = [\alpha']_p$ then $[\beta]_p = [\beta']_p$.

The computational maps of the automaton from finite input sequences to finite output sequences can be generalized as a string function that maps infinite input strings to output strings. Each automaton determines a unique string function describing its computational behavior. Given a string function, it is easy to construct at least one associated automaton merely by letting its internal state be the finite sequence of inputs that it has received.

Definition 2.3. Let F be a string function which maps an input string α to an output string β. We will say that F is a:

(1) *boolean string function* when there is a function G such that for each p, $(\beta)_p = G((\alpha)_p)$.

(2) *finite automata string function* when F is the string function of a finite automaton.

(3) *recursive string function* when there is a recursive function G such that $(\beta)_p = G([\alpha]_p)$.

Here the methods of [3, Chap. 5] can be used to obtain a formal definition of the recursive function G with the specified domain and range. It is obvious that boolean string functions are a proper subset of the finite automata string functions, which in turn are a proper subset of the recursive string functions. For an infinite string α there is a substring of the "foreground" relevant data which may be either finite or infinite. It is useful to have a couple of notations to describe such a substring.

DEFINITION 2.4. Let α and β be strings and p a time:

(1) $\mathscr{R}(\alpha)$, called the *relevant substring* of α, is the result of deleting all occurrences of the blank symbol "b," while preserving the order of the remaining symbols. We will abbreviate $\mathscr{R}(\alpha)$ as $\tilde{\alpha}$.

(2) The strings α and β are said to be *relevance equivalent*, $\alpha \approx \beta$, when $\tilde{\alpha} = \tilde{\beta}$.

(3) The string β is an *alternate extension* of α at p, $\alpha = [p]\beta$, when $[\alpha]_p = [\beta]_p$.

(4) The string β is the result of *inserting a symbol* z into a string α at a time p, $\beta = \alpha\{z//p\}$, when $(\beta)_{p'} = (\alpha)_{p'}$ for $p' < p$, $(\beta)_{p'} = (\alpha)_{p'-1}$ for $p' > p$, while $(\beta)_p = z$.

According to these notations, if $\beta = F(\alpha)$, then $\mathscr{R}(F(\alpha)) = \tilde{\beta}$. The operation of inserting a symbol is complementary to the operation of forming substrings. It follows from this definition that $\alpha = [p-1]\ \alpha\{b//p\}$ and that $\alpha \approx \alpha\{b//p\}$. The difference between the two strings is that after time p, $\alpha\{b//p\}$ contains its relevant data at one time later than α does.

Roughly speaking, we want to say that an automaton is in an "essential blank state" when it is not actually reading its input, so that if a relevant input were provided that automaton would eventually misbehave because it had not read the input. However, we want to characterize this concept as a property of the string function describing the behavior of the automaton. Suppose that at a time p the automaton has been provided a finite sequence of inputs $[\alpha]_p$, where $(\alpha)_p = b$. We want to say that this blank is "essential" when, for some way of further extending inputs through later times, the eventual relevant output would have been different had the blank not been given, allowing each further relevant input to arrive at an earlier time.

DEFINITION 2.5. Let F be a string function and α be some string in its domain, where $(\alpha)_p = b$ at a time p:

(1) We say that b *occurs essentially* for F with input α at time p when there is a β such that $\beta = [p]\ \alpha$ and a β' such that $\beta = \beta'\{b//p\}$, where $F(\beta') \not\approx F(\beta)$.

(2) An automaton is in an *essential blank state* when an input blank that is provided when it is in such a state would occur essentially for the associated string function.

It is an immediate consequence of this definition that, when blanks occur at both times p and $p+1$, then if the blank occurs essentially at $p+1$, it must also occur essentially at p. The *depth of essential blanks* following p is the number of essential blanks in a block of blanks immediately following p when a sufficient number of blanks are inserted following p to include non-essential blanks. Were it possible to insert arbitrarily many essential blanks, we would say that the depth was infinite. We will show that this is not possible for the foreground automata defined below.

Our concept of an essential blank state has been defined for any string function or automaton. Our goal is to define a class of "foreground automata" which are roughly describable as "ignoring the blank inputs." The problem here is that an essential blank input cannot be ignored in the sense that the next relevant input could have been given instead. Only inessential blanks are "truly ignored" in this way. A foreground automaton should need a blank input to be essential solely because it is not responding to input at that time, and the output should be exactly the same no matter what input had been given at that time.

DEFINITION 2.6. A *foreground string function* is a string function that satisfies the condition that for any input string α and time p, if $(\alpha)_p = b$ is an essential blank at p and β is an input string differing from α only at time p, then $F(\alpha) = F(\beta)$. A *foreground automaton* is an automaton with a foreground string function.

In other words, if a foreground automaton ever receives a blank that could not be deleted without affecting some future relevant output, then, in effect, the automaton was not reading its input at that time.

PROPOSITION 2.1. *Let F be a foreground string function and α an input string for F such that at time p, $(\alpha)_p = b$ is not an essential blank. Let $\alpha' = \alpha\{b//q\}$ for some $q < p$. If the resulting blank in α' at time $p+1$ is now an essential blank in α', then the blank at q is also essential.*

Proof. Since the resulting blank in α' at $p+1$ is essential, there is an alternate extension β at $p+1$ such that if β' is the result of deleting the blank at $p+1$, then $\mathscr{R}(F(\beta)) \neq \mathscr{R}(F(\beta'))$. Let δ and δ' be the respective results of deleting the blank at q in β and β'. Let δ and δ' be the respective results of deleting the blank at q in β and β'. Assume that the inserted blank at q in α' was not essential. Now β and β' are both alternate extensions of α' at q so $\mathscr{R}(F(\beta)) = \mathscr{R}(F(\delta))$ and $\mathscr{R}(F(\beta')) = \mathscr{R}(F(\delta'))$. On the other hand, δ is an alternate extension of α at p, with δ' the result of removing this blank. Since this blank was not essential, $\mathscr{R}(F(\delta)) = \mathscr{R}(F(\delta'))$, contradicting the above inequality. ∎

DEFINITION 2.7. Let F be a foreground string function and α an input string for F:

(1) The string α is *proper* for F if there is no time p such that if $\alpha' = \alpha\{b//p\}$ then the blank at p in α' is essential.

(2) The string α is a *minimal input sequence* for F when it is proper for F and if α has an inessential blank at a time p there is no $q > p$ such that $(\alpha)_q \neq b$; i.e., the only way that inessential blanks can occur are in an infinite terminal sequence.

(3) *The relevant output has been determined* at p for input α when for any input string β such that $[\beta]_p = [\alpha]_p$, $F(\beta) \approx F(\alpha)$.

When the relevant output has been determined, further relevant inputs will have no effect on the substring of relevant outputs that eventually emerge. However, they might still have an effect on the times that the outputs appear.

PROPOSITION 2.2. *There is no foreground string function F, input string α for F, and time p such that the depth of essential blanks following p is infinite.*

Proof. Otherwise, if β is any input string for F which is an alternate extension of $[\alpha]_p$, then we can show that for each q, $[F(\alpha)]_q \approx [F(\beta)]_q$ by repeated applications of the definition of a foreground string function. This allows that the relevant output is determined at p so that a blank at p is not essential. ▌

THEOREM 2.1. *Let F be a foreground string function and α an input string for F:*

(1) *There is a unique input string β where $\beta \approx \alpha$ and β is proper and minimal for F.*

(2) *If α is proper for F and a non-essential blank occurs at a time p in α and α' is the result of deleting the blank at p, then α' is again proper for F. More generally, the result of deleting all non-essential blanks which are not in a terminal sequence is a minimal input string for F.*

(3) *If α is proper for F, then the result of inserting a blank at any time q is again proper for F.*

(4) *If α and β are both proper input strings for F and $\alpha \approx \beta$, then $F(\alpha) \approx F(\beta)$.*

Proof. For (1) begin with the string $\tilde{\alpha}$ and look for the first place, if any, that the depth of essential blanks is not zero and insert a number of blanks equal to this depth, which is finite by Proposition 2.2. If $\tilde{\alpha}$ is finite this process will terminate after a finite time and we can append a terminal sequence of non-essential blanks to obtain β. Otherwise β is the result of the non-terminating application of this process.

For (2) we suppose that α' was not proper so that for some $q > p$ and some alternate extension β', where $[\beta']_q = [\alpha']_q$, if we define $\delta' = \beta'\{b//q\}$ then $F(\delta') \not\approx F(\beta')$. Define $\delta'' = \delta'\{b//q\}$ and $\alpha'' = \alpha\{b//q+1\}$ so that by the construction $[\delta'']_{q+1} = [\alpha'']_{q+1}$. Since α is proper the blanks at p and $q+1$ are non-essential in α'', which shows that they are also non-essential in δ''. Now define β as the result of deleting the blank at $q+1$ in δ''. By the construction, $\beta = \beta'\{b//p\}$, which is an alternate extension of $[\alpha]_p$. We can now conclude the following:

(1) $F(\beta) \approx F(\beta')$ because the blank at p is non-essential in α and β is an alternate extension of $[\alpha]_p$.

(2) $F(\beta) \approx F(\delta'')$ because the blank at $q+1$ is non-essential in δ''.

(3) $F(\delta') \approx F(\delta'')$ because the blank at p is non-essential in δ''.

Together, these contradict the above claim that $F(\delta') \not\approx F(\beta')$, thus showing that α' must be proper. We can now systematically delete all non-essential blanks which are not in a terminal sequence with a resulting sequence that remains proper and is minimal.

For (3) let $\alpha' = \alpha\{b//q\}$. Suppose that α' were not proper so that for some $p > q$, $\alpha'' = \alpha\{b//p\}$, and this blank at p is essential. Let β be the result of deleting the blank at q from α''. By construction, $\beta = \alpha\{b//p-1\}$ and is thus a non-essential blank. But by Proposition 2.1 this blank cannot become essential as the result of inserting a non-essential blank at q.

For (4) let δ be the minimal input string constructed from (1) above so that $\delta \approx \alpha$. It is now possible to systematically insert blanks into δ to arrive at a string δ' which can be obtained from both α and β by inserting blanks. By (3) δ' is proper and all of the inserted blanks are non-essential. By (2) these non-essential blanks may be deleted in any order with the resulting strings remaining proper and the corresponding non-essential blanks remaining non-essential. By this construction we can obtain both α and β by inserting non-essential blanks in δ. Hence $F(\alpha) \approx F(\delta)$ and $F(\beta) \approx F(\delta)$. ∎

Part (4) of Theorem 2.1 establishes that a foreground string function F has a well-defined operation that maps the "foreground" relevant substring of a proper input string α for F onto the "foreground" relevant substring of the output. Furthermore, for any input string α, there is a proper minimal string β such that $\beta \approx \alpha$, where any proper string δ such that $\delta \approx \alpha$ can be obtained from β by the insertion of non-essential blanks. This allows us to introduce the following definitions.

DEFINITION 2.8. Let F, F_1, and F_2 be foreground string functions with a common domain and α an input string in this domain:

(1) The string α is *relevance-led* when it has no blank followed by a relevant symbol. Such a string either has no blanks or else it has only finitely many relevant symbols followed by a terminal sequence of blanks.

(2) F^*, the *foreground operation* of F, is a function whose domain is the set of relevance-led input strings for F and whose range is included in the relevance-led output strings. When α is a relevance-led input string for F and β is the minimal

input string such that $\beta \approx \alpha$, $F^*(\alpha) = \mathscr{R}(F(\beta))$ with an appended infinite terminal sequence of blanks in the case that $\mathscr{R}(F(\beta))$ is finite.

(3) F_1 and F_2 are said to be *operationally equivalent*, $F_1 \approx F_2$, when $F_1^* = F_2^*$.

It should be noted that F^* is not necessarily a string function and that operational equivalence is an equivalence relation.

Our simplifying assumption that the "background" is represented by a single symbol "b," which appears in both the input and output vocabularies, could easily be generalized to allow sets of "background" symbols. Such a generalization can prove to be particularly useful in the study of cascades of automata, such as found in [1, Chap. 8]. Suppose that we have an input vocabulary X with some subset B that we want to treat as the set of background symbols. There is a one-state *blanking automaton* that maps all elements of B to the blank symbol "b," while mapping members of $X - B$ onto themselves. To allow B to be viewed as the input blanks for a foreground automaton M, we would require that there be a foreground automaton M' meeting our previous criteria such that the string function of M is equal to the string function for the series cascade of M' following the blanking automaton for B. To also allow some subset B of the output vocabulary to be viewed as blanks, we would require that the series cascade of M followed by the blanking automaton for B be a foreground automaton. Unlike the previous theory, this generalized concept of a foreground automaton restricts the allowed treatment of output blanks as well as input blanks.

3. Computation Time for Foreground Operations

Once we have made a distinction between the foreground and the background and we have a well-defined "foreground operation" for a foreground automaton, one can compare the computational efficiency of operationally equivalent automata. As a benchmark for such a comparison we seek a "maximally efficient" automaton that has no essential blank states and always produces a relevant output when such an output is determined.

Definition 3.1. Let F and F' be a foreground string functions:

(1) F is *maximally efficient* when for every string function F'' such that $F'' \approx F$, and every input string α which is minimal for F, if β is proper input string for F'' and $\beta \approx \alpha$ then for every time p, $\mathscr{R}[F(\alpha)]_p$ is at least as long as $\mathscr{R}[F''(\beta)]_p$.

(2) F' is a *benchmark* for F when $F' \approx F$, F' is maximally efficient, and all input strings are proper for F'.

There are several alternatives that might be suggested for measuring the "computational time" that a foreground automaton has taken at a given time p when it has received a finite input $[\alpha]_p$ and has produced a certain relevant output sequence as a result. Here we propose comparing the time that it took F to produce its relevant output against the time that the benchmark would have taken. At those times when the automaton produces a blank even though a new output has been determined, we find out how much earlier the benchmark automaton would have been able to produce this new relevant output. This ensures that computational time never increases by more than one in a unit time. There is no sudden surge in computational time when the automaton eventually does produce this determined relevant output. The following definition makes these concepts precise. Proposition 3.1 will establish the existence of the benchmark referred to in part (5) of the definition.

DEFINITION 3.2. Let F be a foreground string function, α a proper input string for F, and $p > 0$:

(1) The *relevant output* determined by F at p with input α is $\mathscr{R}([F(\alpha)]_q)$ for the largest number q such that whenever β is a relevance-led input such that $[\beta]_p \approx [\alpha]_p$, then $\mathscr{R}([F(\beta)]) = \mathscr{R}([F(\alpha)])$, or the infinite string, $\mathscr{R}(F(\alpha))$, when there is no such q.

(2) The *remaining output* required with input $[\alpha]_p$ is the remaining sequence in the determined relevant output that follows $\mathscr{R}([F(\alpha)]_p)$. When the length of the remaining output is non-zero, we say that *a new output is determined* for F.

(3) The *desired output value* for F at p is the first value of the remaining output if a new output is determined, or b otherwise.

(4) The *completion time* for F at p with input α is the largest value $q \leqslant p$ such that a new output is determined for F at q.

(5) Let F' be the benchmark for F and β be a relevance-led string such that $\beta \approx \alpha$. The *computational time* taken by F at p with input α is $c - m$, where c is the completion time for F and m is the least value q such that $\mathscr{R}([F'(\beta)]_q) \approx \delta$, where δ the finite sequence $[F(\alpha)]_{p-1}$ followed by the desired output value for F at p.

The proof of the following proposition is quite straightforward. Define the benchmark function as always producing a relevant output when a new output is determined.

PROPOSITION 3.1. *For any foreground string function F, there is a foreground string function F' which is a benchmark for F.*

Note that if F had not taken computation time prior to p, but produced a blank output at p when a relevant output had been determined, then the computational time caused by this blank would first appear at time $p+1$. Computational time is the result of delays in the emergence of the relevant outputs caused by blanks in the input or output strings. However, not every blank leads to such a "computational delay." The computational time can only be increased at those times that an output blank is produced. If a new output was determined for F at a time p when $(F(\alpha))_p$ was blank, then the computational time will increase and we identify this blank output as an *output delay* at p. If no new output was determined for F at p, but the computational time increased anyway, then there must have been a previous input blank in α that prevented F from receiving sufficient relevant input to determine a new output. In this case we identify the blank output as an *input delay* at p. It is possible for the computational time to decrease at a time p that a relevant output is produced by F; we then say that there was a *delay reduction* at p. Such a delay reduction can only occur when several inputs are required to determine an output, such as with the "comparing automaton" discussed in Section 4 below. A delay reduction can only compensate for previous output delays; input delays create permanent computational time. Each such delay or delay reduction changes the computational time by a value of one. We can define the *accumulated output delay*, the *accumulated input delay*, and the *accumulated delay reduction* at a time p as the sum of the previous times in which such a delay or delay reduction took place. The *actual output delay* at p is the result of subtracting the accumulated delay reduction from the accumulated output delay. This discussion is summarized by the following theorem, whose proof is quite straightforward.

THEOREM 3.1. *Let F be a foreground string function and α a proper input string for F and $p > 0$:*

(1) *The actual output delay is never negative; i.e., the accumulated delay reduction at p can never exceed the accumulated output delay.*

(2) *The computational time at p is the sum of the actual output delay and the accumulated input delay.*

(3) *The computational time at p never exceeds the number of blanks in $[F(\alpha)]_p$ and the accumulated input delay never exceeds the number of blanks in $[\alpha]_p$.*

4. Computational Time for Finite Automata

We now seek to see how computational time might arise in finite automata. The simplest such automaton is a one-state or "boolean" automaton with a boolean string function. With the exception of the trivial case, where there is only one relevant symbol in the output vocabulary, a foreground boolean string function must always produce a blank output when it has a blank input. All input sequences are proper, and blank outputs from relevant inputs will not cause delays. Computational time is merely the sum of blanks in the input string.

Among the remaining finite automata, the simplest is a *unit delay automaton* whose output vocabulary is identical to its input vocabulary. At the time $p = 0$ the output is b; for $p > 0$ the output is identical to the input at the time $p - 1$. Note that the next-state and current-output functions for a finite automaton are boolean string functions whose input are ordered pairs consisting of the current state and the current external input for the finite automata. It is easy to show that a finite automaton can always be constructed as a cascade of these two boolean automata with a unit delay automaton. The delay automaton would follow the current state automaton and send its output back to the two boolean automata as the current state component of their inputs. This construction graphically illustrates the differences between the computational time of a foreground finite automaton and such concepts as the time needed for a signal to pass through a device or the time needed for a sequence of computational processes. It also illustrates the fact that the computational time of a cascade of automata often has little relation to computational times of the component automata in the cascade. The use of cascades can be a practical technique for the construction of computational devices. Here it should be observed that computational efficiency is a global effect of the construction and it often cannot be decomposed into any concept of computational efficiency for the component automata.

There are two quite simple finite automata with input and output vocabularies $\{0, 1, b\}$ that provide useful illustrations. A *comparing automaton* compares an even-numbered relevant input with the previous relevant input and produces the output "1" if they were alike or "0" otherwise. A *doubling automaton* duplicates each relevant input in the output. Thus an initial relevant input beginning "101..." would eventually result in a relevant output beginning "110011...." Both operations can be performed by finite foreground automata that operate without using computational time. It is possible to construct a finite foreground comparing automaton for which all input strings are proper and no input blanks are required. Such an automaton must produce a blank output whenever an odd-numbered relevant input is received. A finite doubling automaton must receive approximately as many input blanks as relevant inputs in order to limit the length of the string that it has to remember. Input strings without blanks cannot be proper for such an automaton.

It is now easy to use these two automata to construct a finite foreground automaton whose input vocabulary contains an additional special symbol "#." It begins by operating as a doubling automaton until, if ever, it receives a "#" as its input. Thereafter, it operates as a comparing automaton. Such an automaton must receive input blanks, which do not contribute to computational time as long as it continues to operate as a doubling automaton. However, once it switches to operating as a comparing automaton, these input blanks will have prevented it from receiving sufficient input to determine the output. It will then be forced to produce output blanks which will show up as input delays. Any finite automaton that performs such an operation must sometimes use computational time, but this time arises only from input delays. In the theorem below we will show that a finite foreground string operation can always be performed without output delay.

THEOREM 4.1. *Every finite foreground string function is operationally equivalent to a finite foreground string function that never produces output delay.*

Proof. We will begin with an arbitrary finite foreground automaton M and show how to construct an operationally equivalent finite foreground automaton which produces no output delay. We can assume that M has an initial state that cannot be entered from any other state and that otherwise it has a minimal number of states in that all remaining states are pairwise distinguishable by their eventual functional behavior on some input sequence. The future behavior of an automaton following a time p is determined by the input x and the state q of the automaton at p. Thus the question of whether an output blank at p is an output delay depends on the values of x and q. When there is such an x, we say that q is an *output delay state*. A state q of M will be called an *initial output delay state* when there is some input x such that q gives an output delay with input x, but there is no q' and x' such that M in state q' with input x' gives an output delay and then goes to state q.

LEMMA. *If M has an output delay state, then it has an initial output delay state.*

Proof. We first note that if an output delay occurs at p with a proper input string α, then some relevant output was determined at p. If β is any proper input sequence such that $[\beta]_p = [\alpha]_p$, then there is some $q > p$, where this relevant output is produced at q with input $[\beta]_p$, but the outputs are blank between the times p and q.

Were the lemma false, we could begin with some output delay state q with an input x and indefinitely trace backwards through earlier states, noting for each earlier time a proper input that produces output delay.

Since the automaton is finite, we must finally come to an output delay state q' reading an input x' that can be traced backwards to q' again. Turning this backwards path of inputs around, we obtain a finite proper input sequence which, when given to the automaton in state q', will end up with the automaton again in the state q' while producing only blank output during the process. Repeating this sequence indefinitely, we would obtain an infinite input string for which the automaton would produce only blank output. Since no further relevant output would be produced by this string, it follows that a relevant output was not determined and there could not have been an output delay. ▮

Each state q of the automaton determines a sequence of relevant outputs which will be eventually produced no matter what input is given. The length of this determined relevant output can vary from zero to infinite. We define *r-length*(q), the *relevance length* of the state q, to be n when the length of the determined relevant output is n, where this length is neither zero nor infinite. Otherwise r-length$(q) = 0$. Let r be the maximum of r-length(q) for all initial output delay states q of M. Let s be the number of initial output delay states q such that r-length$(q) = r$. Define the *ordinal of M* to be the ordinal number $r\omega + s$. We complete the proof of the theorem by showing that if the automaton produces an output delay, then we can modify it to create an operationally equivalent finite foreground automaton with a smaller ordinal. Since there can be no infinite descending sequence of ordinal numbers, this construction must terminate with an automaton without output delays.

If the automaton has a state q in which all future relevant output is determined, it is quite easy to show that we can alter the automaton to create an operationally equivalent automaton in which this output is produced as a relevance-led sequence. This would be done by creating a series of new states to follow q. This produces an operationally equivalent automaton that produces output delays only in those states at which the entire future relevant output has not yet been determined. The ordinal of the automaton would be unchanged by this modification.

We now assume that the automaton has an output delay and that the ordinal is greater than zero. Let q be some initial output delay state with the maximal r-length. Assume that the first relevant output determined at q is the symbol "1." For each proper input string β that might be provided to the automaton once it is in state q, there is a number m such that after starting the automaton in state q and giving it the input sequence β, the relevant output of "1" first occurs in the sequence $[\beta]_m$. We will call $[\beta]_m$ a *generating input for the state* q. There are finitely many distinct generating inputs for the state q and they form a tree under the relation of sequence extension.

Now modify the automaton so that in the state q the modified automaton produces a "1" as a result of any input and then goes through a cycle of newly created states giving output blanks until a generating input for the previous automaton has been received. The modified automaton then produces another output blank instead of the "1" that the original automaton would have produced. After this it returns to whatever state of the old automaton that it would have been in after receiving this generating input. This modified automaton is operationally equivalent to the original, but q and all newly created states have a relevance length less than r. Therefore, the original of the modified automaton must be less than that of the original. After a finite number of such modifications, the ordinal must be reduced to zero, creating an operationally equivalent automaton with no output delay. ▮

One interesting open problem is that of obtaining a characterization of those finite automata that are operationally equivalent to a delay-free finite automaton. Many commonly encountered finite automata need not in principle use any computational time. For many practical devices, computational time arises not from the logical necessity of input delay, but from the complementary nature of computational time and internal complexity. Often computational times can be reduced only at the expense of a great increase in the internal complexity. For example, there is a mere boolean automaton capable of playing a perfect chess game with no computational time.

5. Recursive String Functions

We have said that a string function F mapping an input string α to an output string β is "recursive" when there is a recursive function G such that for each p, $(\beta)_p = G([\alpha]_p)$. It is certainly not possible for every recursive string function to be generated by an automaton that could be described as "effective" or "finitely realizable." No provision is made for the computational time that might be needed to compute G. However, we shall see that a recursive foreground string function is operationally equivalent to the string function of such an effective automaton. Thus the computation described by the operation of a recursive foreground string function is appropriately effective even though the string function itself may not have that property of finite realizability that we associate with the term "recursive."

The obvious choice for a class of "effective" automata that extend beyond finite automata is provided by the concept of a Turing machine. Our discussion of Turing machines will be informal and we presume that the reader is familiar with various formal representations, such as is found in [2, Chap. 2]. Turing machines can have multiple tapes, and it is possible for each tape to have a separate vocabulary, multiple heads, and even to be multi-dimensional. The straightforward way to link our theory of automata to Turing machines is to model an automaton by a multi-tape Turing machine which has one-way "input" and "output" tapes, each with a single head that always moves to the right, and where the machine always writes to the output tape. In this case the input and output tapes play the role of the inputs and outputs of the automaton. When such a machine has no additional tapes it can only model a finite automaton, but with additional tapes its computational capabilities are greatly extended. We will call any such automaton that is modeled by such a Turing machine a *Turing automaton*. We will identify and number the "tapes" of a Turing automaton as those tapes other than the input and output tapes. Thus a "one-tape Turing automaton" has one tape in addition to the input and output tapes. The obvious advantage of a Turing automata model is that it simultaneously satisfies the standard definition of a Turing machine and the definition of automata studied in this paper.

THEOREM 5.1. *Every foreground recursive string function is operationally equivalent to the string function of a one-tape Turing automaton.*

Proof. To prove this we need merely note that there is a one-tape Turing machine which, when given a tape with $[\alpha]_p$ as the symbols will eventually halt in a state that encodes $(\beta)_p = G([a]_p)$. Since F is a foreground string function, this tape need only contain the relevant substring of $[\alpha]_p$. It is a straightforward task to modify this one-tape Turing machine into our desired Turing automaton. For example, this modified machine might keep the relevant substring of $[\alpha]_p$ as an initial string on its third tape, where this initial string was followed by a special marking symbol not in the input or output vocabulary. The computations could then be carried on in the remainder of the tape. Each time that the Turing automaton received a new relevant input, it would begin by appending this input to the initial segment of its tape. A proper input string would have to provide essential blank inputs until this rewriting was completed and the machine had either determined a new relevant output or determined that additional relevant input was needed. ∎

Such a machine would be quite computationally inefficient. However, some inefficiency is unavoidable and a Turing automaton with only one tape and tape head often cannot avoid essential input blanks. Since a Turing automaton can have only finitely many internal states and has a finite tape vocabulary, the only way that it could store a continuous stream of new relevant inputs would be to continuously write these on its tape. But this would prevent it from using the tape for any other purpose and it would encounter the same problems as would a finite automaton. A Turing automaton with added tapes and heads can avoid this problem and take all input strings as proper. In contrast to the finite automata that can only avoid output delay, these expanded Turing automata can always avoid input delay, but might still have output delays.

Further investigations into the computational times of Turing automata can be carried out by applying previous results on the computational complexity of Turing machines. One immediate application is to show that there are Turing foreground automata for which no Turing automaton can serve as a benchmark. It is easy to apply the "speedup theorems," such as [2, Theorem 12.14], to show that there are foreground operations for which every Turing automaton could be replaced by a yet more efficient Turing automaton and none could serve as a benchmark.

Theorem 5.1 establishes that a form of "Church's Thesis" can be applied to our concept of a foreground string operation; i.e., every foreground string operation that can be computed by an effective procedure can be performed by a one-tape Turing automaton. Quite a different picture emerges when we inquire which foreground string functions can be performed by an automaton which might be described as carrying out an "effective procedure." Standard results from the theory of computational complexity show that the class of Turing automata expands as these automata are allowed more tapes and heads. It is an interesting question whether there is some wider class of automata for which it would be reasonable to propose such analogue of Church's Thesis. For example, we might propose a class of Turing automata with arbitrarily large finite numbers of tapes, heads and dimensions for each tape. However, I would speculate that no such proposal would gain the widespread acceptance that has been accorded to Church's Thesis. A very generalized concept of a Turing machine is merely a finite automaton which is allowed to read and write on a "discrete world," which is assumed to satisfy certain criteria. However, it is not clear what limitations should be placed on such criteria. For example, one such world might consist of a multi-dimensional tape with highly non-Euclidean topological properties that allows a tape head to move n cells away in one of the dimensions by selecting a path through other dimensions, where the selected path forms a binary coding of the number n. It would then be able to move n cells away with only $\log(n)$ moves. Such an automaton could have computational capabilities that would not be available to an automaton whose tapes were restricted to a Euclidean geometry. No such contorted tape could be constructed in our Euclidean world, but this would seem to be a mere empirical restriction, not a logical one.

For previous automata that we have considered, including the Turing automaton constructed for the proof of Theorem 5.1, it was always easy to determine when the automaton was in an essential blank state. Our general theory has not taken into account those methods by which a symbiosis might be achieved between the physical device modeled by an automaton M that requires essential blanks in its input and an "external world" that cooperates by providing a proper input to the device. One possibility might be for the physical device to embody a second automaton whose output signals the necessity of an essential blank for M. We call such an automaton the *input-seeking automaton* for M. For a finite automaton there is no computational complexity in deciding whether the next state is an essential blank state. The subset of essential blank states can be determined at the time that the automaton is constructed. On the other hand, a Turing automaton has a potential infinity of "states" determined in part by the status of the additional internal tapes used for computation. In this case it is not always decidable whether or not the automaton is in an essential blank state and we may not be able to construct a recursive input-seeking automaton.

THEOREM 5.2. *There is a foreground Turing automaton for which the question of whether or not the automaton is in an essential blank state is undecidable.*

Proof. To show how to construct such an automaton, we first note that it is easy to construct a two-head one-tape Turing automaton with no essential blank states, which performs the doubling operation described above. The first head writes the inputs on the tape while the second follows behind and reads them at the appropriate times. Next, we use the techniques of [3, Chap. 1] to construct a second Turing machine that, when given a tape with n 1's followed by a 0 halts if and only if the nth recursively enumerable set eventually enumerates the number one. To construct the desired automaton, we begin with the doubling Turing automaton and add a second tape. As long as only input 1's have been received, the automaton copies these to this tape. If an input 0 is received after a length n initial block of 1's, the head on the second tape proceeds to mimic the second Turing machine. If this Turing machine ever determines that the nth recursively enumerable set enumerates the number one, then the Turing automaton ceases to move the second head that was used for the doubling operation. Thereafter it attempts to double each new relevant input just like the finite doubling automaton described above. Once the number n has been determined for the second Turing machine by the input of a 0, further input blanks will be essential if and only if the nth recursively enumerable set eventually enumerates the number one, an undecidable question as shown in [3, Chap. 2]. ∎

Automata such as the one constructed for Theorem 5.2 seem artificial and undesirable. We might want to rule them out altogether when defining the class of foreground automata. However, the main focus of this paper has been to show that one very simple criterion is sufficient to define a class of foreground automata that determine well-defined string operations. This gives us a simple basic theory that can be the foundation for future refinements.

REFERENCES

1. M. ARBIB, "Theories of Abstract Automata," Prentice-Hall, Englewood Cliffs, NJ, 1969.
2. J. HOPCROFT AND J. ULLMAN, "Introduction to Automata Theory, Languages, and Computation," Addison–Wesley, Reading, MA, 1979.
3. H. ROGERS, JR., "Theory of Recursive Functions and Effective Computability," McGraw–Hill, New York, 1967.