WPI-CS-TR-95-4 December 1995

Self-Modifying Finite Automata — Power and Limitations

by

John N. Shutt

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Self-Modifying Finite Automata — Power and Limitations

John N. Shutt
jshutt@cs.wpi.edu
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

December 1995

Abstract

The Self-Modifying Finite Automaton (SMFA) is a model of computation introduced in [RS93, RS94, RS95b]. Formal definitions appear in [RS95a]. This paper further investigates the computational power of the model, and introduces the concepts of *path determinism* and *register complexity*.

Contents

1 Introduction

The Self-Modifying Finite Automaton (SMFA) is a model of computation introduced in [RS93, RS94, RS95b]. SMFAs are similar to standard finite automata, but changes to the transition set are allowed during a computation. Formal definitions appear in [RS95a]. A weakly restricted form of SMFAs has been shown to be Turing powerful [RS95a], and strongly restricted forms have been shown to accept the class of metalinear languages, as well as some other classes of context-free and even non-context-free languages [RS94, RS95b].

This paper further investigates the computational power of SMFAs. Results presented establish lower or upper bounds on the computational power of various classes of SMFAs. Of particular interest are results concerning the important class of ρ-normal single-addition single-register first-order SMFAs without deletion; the computational power of this class is bounded below by the ultralinear languages, and above by a pumping lemma. Other important developments including the property of *path determinism*, and the consideration of number of registers as a complexity measure.

2 Preliminaries

All the definitions and conventions of [RS95a] are assumed here without repetition. For any alphabet X, $X_\lambda = X \cup \{\lambda\}$.

The following additional definitions and conventions are adopted henceforth.

Convention 2.1 (ρ-normal)
All SMFAs are assumed ρ-normal except where explicitly otherwise stated. □

For example, a theorem that explicitly addresses "SMFAs with self-delete" is understood to concern SMFAs with ρ-normal addition and self-delete (since self-deletion is not ρ-normal).

Convention 2.2 (Without deletion)
All SMFAs are assumed without deletion except where explicitly otherwise stated. □

So a theorem that explicitly addresses "SMFAs" is understood to concern ρ-normal SMFAs without deletion.

Convention 2.3 (Order of SMFAs with self-delete)
When an SMFA is stated to be with self-delete, any statement about the order of the machine refers only to the addition function. Similarly, the order of an action of such a machine is its order in the addition function. □

For example, a "first-order SMFA with self-delete" has a first-order addition function. Self-deletion is unordered, so there is nothing more to be usefully said about the order of the deletion function. The need for this convention did not arise in [RS95a] because SMFAs with self-delete were only peripherally mentioned there.

Note particularly that the above conventions pertain to SMFAs, *not* SMAs. Any definition or result that explicitly addresses "SMAs" entails no implicit assumptions about the properties of modification functions.

Definition 2.4 (Register complexity)
Suppose $\mathcal{C}$ is a set of SMAs, and L is a language accepted by some $M \in \mathcal{C}$. Then the *register complexity* of L (in $\mathcal{C}$) is the smallest number of registers of any $N \in \mathcal{C}$ that accepts L. □

3 Determinism

Among the most universally applicable distinctions between classes of automata is that between deterministic and nondeterministic: Does the machine always have to proceed in a certain way, or is there sometimes more than one way to go?

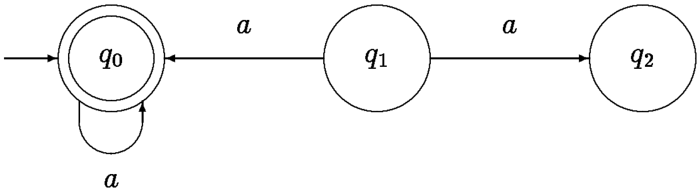
Interestingly, even the *definition* of determinism for SMFAs — conspicuous by its absence from [RS95a] — is not altogether straightforward. An ordinary finite automaton, or even an ordinary Turing machine, is deterministic iff its transition function is single-valued. Since the transition function is both fixed and finite, it is always immediately obvious whether or not the machine is deterministic. But an SMFA doesn't have a transition function per se; and its transition set is neither fixed nor, in general, bounded.

3.1 Basic criteria

One deceptively obvious way to proceed would be to say an SMFA is deterministic iff at each computation step there is at most one transition that can be taken at any point in a computation.

Definition 3.1 (Path determinism)
An SMA M is *path-deterministic* iff for each reachable configuration c of M there is at most one transition allowed from c. □

A shortcoming of path determinism is illustrated by the following machine.



This machine is a finite automaton, accepting the language a^*. And, like all finite automata, it could equally well be considered a zeroth-order SMFA.

As an finite automaton, it is obviously nondeterministic; there are two different transitions on input a from state q_1. But as an SMFA, it is path-deterministic. (One might object that q_1 is a useless state; but in general, the halting problem will make it impossible to decide which states of an SMFA are useless.) This suggests a stronger definition of determinism.

Definition 3.2 (State determinism)

Suppose M is an SMA. A configuration c of M is *state-deterministic* iff for each configuration c' of M with the same transition set as c, there is at most one transition allowed from c'. M is *state-deterministic* iff every reachable configuration of M is state-deterministic. □

In other words, on any given input there is at most one transition allowed from each state of each reachable configuration.

A finite automaton is deterministic if and only if, interpreted as a zeroth-order SMFA, it is state-deterministic. Note that state determinism implies path determinism.

Definition 3.2 imposes determinism on each state of a reachable configuration, but only to a depth of one computation step. This suggests yet another, even stronger criterion, as follows.

Definition 3.3 (Strong state determinism)

Suppose M is an SMA. Let $\triangleright$ be the following binary relation on configurations of M: $c \triangleright c'$ iff there exists a configuration c'' of M with the same transition set and register function as c such that $c'' \vdash_M c'$. A configuration c of M is *strongly state-deterministic* iff for all configurations c' of M, $c \triangleright^* c'$ implies c' is state-deterministic. M is *strongly state-deterministic* iff every initial configuration of M is strongly state-deterministic. □

Note that strong state determinism implies state determinism.

As the following theorem demonstrates, strong state determinism is exceedingly difficult to satisfy; in fact, almost none of the SMFAs in the literature to date have been strongly state-deterministic.

Theorem 3.4 (Strongly state-deterministic SMFAs)
If M is a strongly state-deterministic SMFA, then $L(M)$ is regular. $\Box$

Proof. Suppose M is an SMFA.

Suppose there are only finitely many reachable states (that is, current states of reachable configurations). Between finitely many states, there are only finitely many possible transitions. Therefore, it is possible to construct a finite automaton that simulates M, and thus accepts the same language. So $L(M)$ is regular.

On the other hand, suppose there are an infinite number of reachable states. Then there must be a reachable state of the form $\langle r, k\rangle$ with $k \geq 1$. Let $c_1 = \langle\langle r, k\rangle, \rho, \delta, y\rangle$ be a reachable configuration with current state of this form. Let $c_0 = \langle q_0, \rho_0, \delta_0, xy\rangle$ be an initial configuration of M from which c_1 is reachable.

Let $c'_0 = \langle q_0, \rho, \delta, xy\rangle$, the configuration constructed from c_1 by replacing its current state with the initial state, and rewinding the input. Let c'_1 be the configuration computed from c'_0 by following the same transition path that was used in the computation $c_0 \vdash^*_M c_1$. Traversing this path from c'_0 reads input x, and performs a sequence of actions that *creates* a path from q_0 to state $\langle r, 2k\rangle$. Since $k \geq 1$, $\langle r, 2k\rangle \neq \langle r, k\rangle$. So in the transition set of c'_1 there are two different paths from q_0 that read the same input but end in different states. So c'_1 is not state-deterministic. But c_1 is reachable, c'_0 has the same transition set and register function as c_1, and $c'_0 \triangleright^* c'_1$. So M is not strongly state-deterministic. $\Box$

The above proof relies on Conventions 2.1 and 2.2. The contradiction in the proof stems from the fact that the transitions that created the path to c_1 are still available for reuse. If some or all of these transitions had been self-deleting, strong state determinism might have been maintained. (See §3.3, below.)

3.2 Qualified criteria

It will be of particular interest to note how the imposition of path determinism affects the register complexity of various languages (in various classes). In this regard, there are also several criteria related to determinism, but weaker than those of §3.1 above, whose effect on register complexity may also sometimes be of interest.

Definition 3.5 (Lookahead determinism)
Suppose M is an SMA. A configuration c of M is a *dead end with lookahead 0* iff c is nonaccepting, and there are no transitions allowed from c. c is a *dead end with lookahead* $n+1$ iff c is nonaccepting, and $c \vdash_M c'$ implies c' is a dead end with lookahead n.

M is *lookahead* 0 *path-deterministic* iff M is path-deterministic. M is *lookahead* $n+1$ *path-deterministic* iff for every reachable configuration c of M, there is at most one transition d such that $step(c, d)$ is not a dead end with lookahead n. $\Box$

If c is a dead end with lookahead n, then c is also a dead end with lookahead m for all $m > n$.

No lookahead variant of state determinism will be defined. The obvious formulation of such a variant would suffer from the same anomaly that befalls strong state determinism (Theorem 3.4).

Definition 3.6 (Determinism up to trivial transitions)

Suppose M is an SMA. A transition d over M is *trivial* iff d is a zeroth-order λ-transition. Let $\triangleright$ be the following binary relation on configurations of M: $c \triangleright c'$ iff there exists a trivial transition d such that $step(c, d) = c'$.

A configuration c of M is *path-deterministic up to trivial transitions* iff there is at most one allowable pair $\langle c', d\rangle$ such that $c \triangleright^* c'$ and d is nontrivial. M is *path-deterministic up to trivial transitions* iff every reachable configuration of M is path-deterministic up to trivial transitions.

A configuration c of M is *state-deterministic up to trivial transitions* iff every configuration with the same transition set as c is path-deterministic up to trivial transitions. M is *state-deterministic up to trivial transitions* iff every reachable configuration of M is state-deterministic up to trivial transitions. □

State determinism can be safely varied to allow trivial transitions, where it could not for lookahead, because the anomaly of Theorem 3.4 was caused by reusing transitions that modified the machine; trivial transitions present no such difficulty.

It is also possible to combine lookahead with trivial transitions, in the case of path determinism (though not, of course, state determinism, which has no lookahead variant). The resulting definition is particularly labyrinthine.

Definition 3.7 (Lookahead determinism up to trivial transitions)

Suppose M is an SMA. Let $\triangleright$ be the relation defined in the previous definition.

A configuration c of M is a *dead end with lookahead 0 up to trivial transitions* iff $c \triangleright^* c'$ implies both that c' is nonaccepting and that only trivial transitions are allowed from c'. c is a *dead end with lookahead* $n+1$ *up to trivial transitions* iff $c \triangleright^* c'$ implies both that c' is nonaccepting and that, if d is a nontrivial transition allowed from c', then $step(c', d)$ is a dead end with lookahead n up to trivial transitions.

A configuration c of M is *lookahead* 0 *path-deterministic up to trivial transitions* iff c is path-deterministic up to trivial transitions. c is *lookahead* $n+1$ *path-deterministic up to trivial transitions* iff there is at most one allowable pair $\langle c', d\rangle$ such that $c \triangleright^* c'$, d is nontrivial, and $step(c', d)$ is not a dead end with lookahead n up to trivial transitions.

M is *lookahead* n *path-deterministic up to trivial transitions* iff every reachable configuration of M is lookahead n path-deterministic up to trivial transitions. □

3.3 Turing power

In [RS95a], it was proven that SMFAs are Turing powerful. Specifically, given a deterministic Turing machine M, an SMFA was constructed that simulates M. The register complexity of the construction was not particularly remarked on.

This result is repeated below, with register complexity specifically noted. The proof is also repeated here, because subsequently it will be modified to construct SMFAs with various kinds of determinism.

Theorem 3.8 (SMFAs)
Suppose language L is accepted by a deterministic Turing machine with n states. Then there exists an SMFA with $2n+1$ registers that accepts L. $\square$

Proof. Suppose language L is accepted by a deterministic Turing machine $M = \langle Q, Z, T, \delta, q_0 \rangle$, where

- Q is the set of states.
- Z is the tape alphabet, including the blank symbol $\#$, but not symbols L, R, H.
- $T \subseteq Z - \{\#\}$ is the input alphabet.
- $\delta : Q \times Z \to (Q \times Z \times \{L, R\}) \cup \{H\}$ is the transition function.
- $q_0 \in Q$ is the start state.

Here, L, R, H mean "move left", "move right", and "halt".

Configurations are indexed by nonnegative integers $j \in \mathbb{N}$. Tape cells are indexed by integers $k \in \mathbb{Z}$. Let $z_{j,k} \in Z$ be the symbol at cell k, $p_j \in \mathbb{Z}$ the head position, and q_j the machine state, in configuration j. In the initial configuration, with input string $w = w_1 \cdots w_n$, $w_k \in T$,

$$
\begin{aligned}
p_0 &= 0 \\
z_{0,k} &= \begin{cases} w_k & \text{if } 1 \le k \le n \\ \# & \text{otherwise} \end{cases}
\end{aligned}
$$

An SMFA will now be constructed that simulates M, hence accepts L.

Each configuration j is represented by a path of λ-transitions with actions $a_{j,k}$, for $-j \le k \le n+1+j$, where

$$
a_{j,k} = \begin{cases} \langle q_j, z_{j,k} \rangle & \text{if } k = p_j \\ z_{j,k} & \text{otherwise} \end{cases}
$$

Traversing this path constructs a path representing the next configuration (except for the first and last transitions of the new path, which are constructed by predefined

$$
\begin{array}{llcll}
z: & old_0 & \xrightarrow{\lambda/z} & new_0 & \\
 & old_0 & \xrightarrow{\lambda/\langle q,z\rangle} & new_{q,L} & \forall q \in Q \\
 & old_{q,R} & \xrightarrow{\lambda/\langle q,z\rangle} & new_0 & \forall q \in Q \\
\langle q,z\rangle: & old_0 & \xrightarrow{\lambda/z'} & new_{q',R} & \text{if } \delta(q,z) = \langle q',z',R\rangle \\
 & old_{q',L} & \xrightarrow{\lambda/z'} & new_0 & \text{if } \delta(q,z) = \langle q',z',L\rangle \\
 & old_0 & \xrightarrow{\lambda} & q_f & \text{if } \delta(q,z) = H \\
begin: & q_L & \xrightarrow{\lambda/\#} & new_0 & \\
 & q_L & \xrightarrow{\lambda/\langle q,\#\rangle} & new_{q,L} & \forall q \in Q \\
end: & old_0 & \xrightarrow{\lambda/\#} & q_R & \\
begin': & q_L & \xrightarrow{\lambda/\langle q_0,\#\rangle} & new_0 &
\end{array}
$$

Figure 1: Actions of an SMFA to simulate a DTM

transitions using the special actions *begin* and *end*). The following set of $2|Q|+1$ registers is used in the construction:

$$R \;=\; \{r_0\} \cup \{r_{q,L} \mid q \in Q\} \cup \{r_{q,R} \mid q \in Q\}$$

When constructing configuration j from configuration $j-1$, the transitions for the old and new head positions (p_{j-1} and p_j) are connected through register $r_{q_j,d}$, where d is the direction moved by the head between configurations $j-1$ and j. All other consecutive pairs of transitions are connected through r_0.

In order to guarantee that all of the registers will be updated by every action of the machine, every action adds trivial transitions

$$new_r \xrightarrow{\lambda} new_r \quad \forall r \in R$$

Additional transitions are added by various actions, as shown in Figure 1. Here, q_f is the final state, and q_L, q_R are other predefined states.

The entire SMFA is shown in Figure 2. q_s is the start state, and q_f the final state. During computation, the entire input string must be read while in state q_1; otherwise, the remainder of the input will never be read, and by definition the string will not be accepted. Traversing from q_s to q_2 creates the initial configuration path from q_L to q_R. Thereafter, traversing any loop from q_2 to q_2 creates another configuration path from q_L to q_R. There is no requirement that this loop always use the most recently

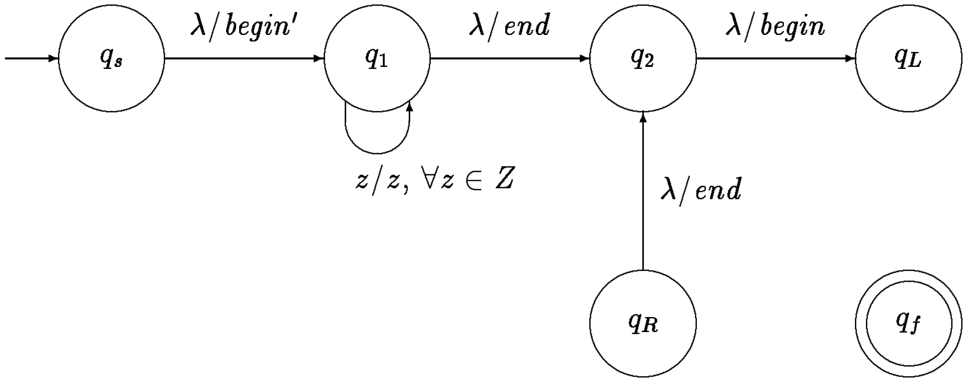


Figure 2: SMFA to simulate a DTM

added configuration path, but repeating an earlier configuration path only creates a redundant copy of some other existing path. The final state is reachable iff some accepting configuration path can be created, iff that configuration is reachable from the given initial configuration. $\square$

The SMFA in the above proof violates determinism (path *or* state) in four ways:

1. There is nothing to keep the machine from leaving state q_1 prematurely, i.e. before all the input has been read.
2. All of the configurations start at q_L, so after the first simulated Turing machine step, each time the SMFA passes through q_L it can take any existing configuration path.
3. Register updates are forced by means of trivial transitions $new_r \xrightarrow{\lambda} new_r$.
4. From each intermediate state within a particular configuration path, there are $n+1$ transitions, all but one of which lead to dead ends.

The first violation is really a consequence of setup rather than any fundamental theoretical shortcoming of SMFAs versus Turing machines: Turing machines have the advantage of a blank symbol marking the end of the input, while SMFAs have no such end marker unless explicitly specified. The difficulty is trivially resolved by a new definition.

Definition 3.9 (Language accepted with end-marker)
Suppose M is an SMA, and L is a language. L is *accepted with end-marker* by M iff $L(M) = L\{\$\}$ for some symbol \$ that does not occur in any string of L. $\square$

The third and fourth violations of determinism in the above theorem are allowed for in the qualified criteria for determinism (§3.2).

The second violation is more serious. It is inimical to every form of determinism considered here. However, it also turns out to be easily eliminated with just two extra registers. (The path-nondeterministic trivial transitions can be eliminated with no extra registers, which saves repetitions of the words "up to trivial transitions".)

Theorem 3.10 (Lookahead 1 path-deterministic SMFAs)

Suppose language L is accepted by a deterministic Turing machine with n states. Then there exists an SMFA with $2n+3$ registers that accepts L with end-marker, and is lookahead 1 path-deterministic up to trivial transitions. □

Proof. The construction is just the same as for Theorem 3.8, except for the specific alterations described below.

The trivial transitions are eliminated by simply replacing each addition

$$new_r \xrightarrow{\lambda} new_r$$

for register r by

$$q_0 \xrightarrow{\lambda} new_r$$

This has the same effect of forcing an update of the register value, and since the start state q_0 is never revisited, it doesn't interfere with path determinism. (State determinism is another matter, though.)

Instead of using predefined states q_R, q_2, and q_L to connect each configuration path to the next, two additional registers are introduced for this purpose, called r_R and r_L.

The two additional registers are not updated by every action (unlike the previously described $2n+1$ registers). The *begin'* action is renamed to *first*; the *begin* and *end* actions are eliminated, and replaced by a new action *next*. The transitions added by *next* are as follows:

$$\begin{array}{rrcll} next: & q_0 & \xrightarrow{\lambda} & new_r & \forall r \in R - \{r_L, r_R\} \\ & old_0 & \xrightarrow{\lambda/\#} & new_{r_R} & \\ & new_{r_R} & \xrightarrow{\lambda/next} & new_{r_L} & \\ & new_{r_L} & \xrightarrow{\lambda/\#} & new_0 & \\ & new_{r_L} & \xrightarrow{\lambda/\langle q,\#\rangle} & new_{q,L} & \forall q \in Q \end{array}$$

The entire SMFA is shown in Figure 3. □

In the SMFA constructions of the preceding theorems, one path was built for each DTM configuration, with transitions incident to each state of r_0 going forward

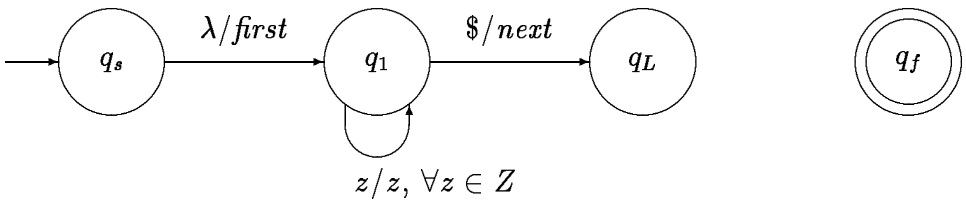


Figure 3: SMFA to simulate a DTM

to states of $r_{q,L}$ and backward to states of $r_{q,R}$. The forward transitions caused the lookahead qualification on path determinism.

The following construction will eliminate this problem by building *two* parallel paths for each DTM configuration: one enumerating the tape contents from left to right, with backward transitions to states of $r_{q,R}$; and one enumerating the tape contents from right to left, with backward transitions to states of $r_{q,L}$.

This considerably complicates the construction. Interestingly, though, the increase in register complexity is fairly modest, because registers of the form $r_{q,d}$ do not have to be duplicated for both paths: registers $r_{q,R}$ are only used by the left-to-right path, and $r_{q,L}$ by the right-to-left.

Theorem 3.11 (Path-deterministic SMFAs)
Suppose language L is accepted by a deterministic Turing machine with n states. Then there exists a path-deterministic SMFA with $2n+6$ registers that accepts L with end-marker. $\square$

Proof. The construction is like that for Theorem 3.10, but with the extensive alterations described below.

Registers that are associated with tape enumeration in just one direction are superscripted with an R (for rightward, i.e. left-to-right) or an L (leftward, right-to-left). Registers r_0 and r_L are each doubled, producing four registers r_0^R, r_0^L, r_L^R, and r_L^L. Registers $r_{q,d}$ are not doubled, but are superscripted; and since the superscript is always identical to the second subscript, the latter is omitted, hence r_q^d. The only registers not superscripted are r_R, and an additional register r_2 (corresponding to state q_2 of the nondeterministic construction). The entire register set is:

$$R \;=\; \{r_R, r_2\} \cup \{r_0^d, r_L^d, r_q^d \mid d \in \{L, R\},\ q \in Q\}$$

The subset of registers that must be incremented by each enumeration step is:

$$R' \;=\; \{r_0^d, r_q^d \mid d \in \{L, R\},\ q \in Q\}$$

Actions associated with tape enumeration are also doubled by direction superscripts, and the addition function handles them differently depending on superscript.

A rightward enumeration builds the next rightward enumeration forward and the next leftward enumeration backward, while a leftward enumeration builds the next *left*ward enumeration forward and the next *right*ward enumeration backward. Action *next* is replaced by actions $begin^d$ and end^d. The only action not superscripted is *first*. The entire action set is:

$$A \;=\; \{\mathit{first}\} \cup \{\mathit{begin}^d, \mathit{end}^d, z^d, \langle q, z\rangle^d \mid d \in \{L, R\},\; z \in Z,\; q \in Q\}$$

R-superscripted registers and actions are only used to construct a *rightward* enumeration of the tape; L-superscripted, only to construct a *leftward* enumeration. The superscripts of the source/destination state registers and action of a transition always agree. The direction of enumeration for every non-initial configuration is the same as the direction of head movement into that configuration from its predecessor; this is what maintains path determinism. For the initial configuration, however, the direction of enumeration is arbitrary; the machine described here enumerates initial configurations rightward, as if in an initial configuration the head has just completed a move right.

During the inception of a new pair of parallel enumeration paths (by an action $begin^d$), registers r_R, r_2, and both r_L^d are incremented. The *old* value of r_R is the termination point for the previous enumeration (the one that is about to be traversed in order to build the new enumerations). This *old* value is connected to the *new* value of r_2; the *new* value of r_R immediately becomes the termination point from which one of the new enumerations grows backward, and to which the other grows forward. The additional register r_2 is needed because the connection to one of the r_L^d cannot be made until the direction of head movement is known, which doesn't happen until the head position of the previous enumeration (action $\langle q, z\rangle^d$) is traversed.

In order to guarantee that all of the registers that *need* to be incremented by every enumeration step *will* be, every action adds trivial transitions

$$q_0 \xrightarrow{\lambda} new_r \quad \forall r \in R'$$

Additional transitions added by the various actions are shown in Figure 4. If d is a direction, that is, $d \in \{L, R\}$, then the opposite direction is denoted $\overline{d} =$ (if $d = L$ then R else L endif).

The entire SMFA is shown in Figure 5. □

The only violation of state determinism in the above path-deterministic construction is that multiple trivial transitions out of q_0 are added to force register update.

Theorem 3.12 (State-deterministic SMFAs)
Suppose language L is accepted by a deterministic Turing machine with n states. Then there exists a state-deterministic SMFA with $2n + 7$ registers that accepts L with end-marker. □

$$
\begin{array}{lrcll}
z^d: & old_0^d & \xrightarrow{\lambda/z^d} & new_0^d & \\
 & old_q^d & \xrightarrow{\lambda/\langle q,z\rangle^d} & new_0^d & \forall q \in Q \\
 & new_0^{\overline{d}} & \xrightarrow{\lambda/z^{\overline{d}}} & old_0^{\overline{d}} & \\
 & new_q^{\overline{d}} & \xrightarrow{\lambda/\langle q,z\rangle^{\overline{d}}} & old_0^{\overline{d}} & \forall q \in Q \\
\langle q,z\rangle^d: & old_0^d & \xrightarrow{\lambda/z'^d} & new_{q'}^d & \\
 & new_0^{\overline{d}} & \xrightarrow{\lambda/z'^{\overline{d}}} & old_{q'}^{\overline{d}} & \\
 & old_2 & \xrightarrow{\lambda/begin^{d'}} & old_L^{d'} & \text{if } \delta(q,z) = \langle q', z', d'\rangle \\
 & old_2 & \xrightarrow{\lambda} & q_f & \text{if } \delta(q,z) = H \\
begin^d: & old_R & \xrightarrow{\lambda/end^d} & new_2 & \\
 & new_L^d & \xrightarrow{\lambda/\#^d} & new_0^d & \\
 & new_0^{\overline{d}} & \xrightarrow{\lambda/\#^{\overline{d}}} & new_R & \\
 & new_q^{\overline{d}} & \xrightarrow{\lambda/\langle q,\#\rangle^{\overline{d}}} & new_R & \forall q \in Q \\
 & q_0 & \xrightarrow{\lambda} & new_L^{\overline{d}} & \\
end^d: & old_0^d & \xrightarrow{\lambda/\#^d} & old_R & \\
 & old_{q,d}^d & \xrightarrow{\lambda/\langle q,\#\rangle^d} & old_R & \forall q \in Q \\
 & old_L^{\overline{d}} & \xrightarrow{\lambda/\#^{\overline{d}}} & old_0^{\overline{d}} & \\
first: & q_2 & \xrightarrow{\lambda/begin^R} & new_L^R & \\
 & new_L^R & \xrightarrow{\lambda/\langle q_0,\#\rangle^R} & new_0^R & \\
 & new_0^L & \xrightarrow{\lambda/\#^L} & new_R & \\
 & new_q^L & \xrightarrow{\lambda/\langle q,\#\rangle^L} & new_R & \forall q \in Q \\
 & q_0 & \xrightarrow{\lambda} & new_L^L &
\end{array}
$$

Figure 4: Actions of an SMFA to simulate a DTM

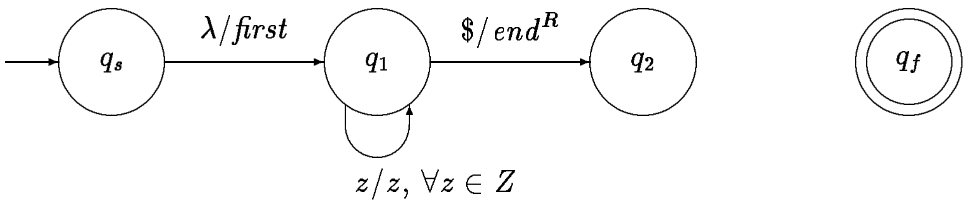


Figure 5: SMFA to simulate a DTM

Proof. The construction is just the same as for Theorem 3.11, except for the specific alterations described below.

Introduce one further register r_x; and for each register $r \in R$ that needs to be forcibly incremented, introduce a new symbol σ_r into the alphabet.

Whenever the previous construction calls for the addition of a transition

$$q_0 \xrightarrow{\lambda} new_r$$

add instead a transition

$$new_x \xrightarrow{\sigma_r} new_r$$

Instead of an unbounded number of trivial transitions from q_0, the new machine has a bounded number of transitions from each $\langle r_x, k\rangle$; and each of these transitions is labeled with a different input symbol, so state determinism is maintained. □

It was shown by Theorem 3.4 that strong state determinism severely over-constrains SMFAs. There may be a way for strongly state-deterministic SMFAs *with self-delete* to achieve Turing power; however, since strong state determinism is of only minimal interest, and constructing strongly state-deterministic machines is very exacting work, the proof is left to the student.

Conjecture 3.13 (Strongly state-deterministic SMFAs with self-delete)
Suppose language L is accepted by a deterministic Turing machine with n states. Then there exists a strongly state-deterministic SMFA with self-delete and $2n + 7$ registers that accepts L with end-marker. □

The concept of state determinism (let along strong state determinism) will not be carried further in this paper.

Convention 3.14 (Path determinism)
Hereafter to the end of the paper, *determinism* without the prefix qualifier "state" will always mean "path determinism", and similarly for the variants of path determinism. □

So a "lookahead 1 deterministic SMFA" is lookahead 1 path-deterministic, etc.

4 SMFAs without λ-transitions

Theorem 4.1 (Deterministic SMFAs without λ-transitions)
If M is a deterministic SMFA without λ-transitions, then $L(M)$ is deterministic context-sensitive. $\square$

Proof. Suppose M is a deterministic SMFA without λ-transitions. It will be shown that $L(M)$ is accepted by some deterministic Turing machine N in linear space.

Throughout the construction, u_k denotes the k^{th} symbol in a string u, and for $j \leq k$, $u_{(j,k)}$ denotes the substring of u from u_j to u_k inclusive. (Thus, $u_{(k,k)} = u_k$.) For $j > k$, $u_{(j,k)} = \lambda$.

Let the tape alphabet of N consist of the input alphabet, predefined states, registers, and actions of M, and a blank symbol. One can always use disjoint union to force all these symbols to be distinct; so assume distinctness WLOG.

Suppose v is a string of actions of M, and $c = \langle q, \rho, \delta, w\rangle$ is a configuration of M that can be reached (from some initial configuration) while performing actions v. If q is a predefined state, then c will be represented in N by the string vqw. If $q = \langle r, k\rangle$, then c will be represented in N by a string $v_{(1,j)}rv_{(j+1,|v|)}w$ for some $0 \leq j \leq |v|$, where k was the current value of register r after the j^{th} step of the computation.

When c is not an accepting configuration, machine N uses the algorithm shown in Figure 6 to find a transition of M allowable from c. The result of the algorithm consists of the transition $q \xrightarrow{s/a} p$ to be taken (using ρ-relative notation for created states), and an integer k indicating the computation step at which the transition was added. If the transition is in the initial transition set of M, $k = 0$; otherwise, the transition was added by action v_k.

Given transition $d = (q \xrightarrow{w_1/a} p)$ and integer k returned by the algorithm of Figure 6, machine N computes $step(c, d)$ as follows.

if p is
  predefined: return $vapw_{(2,|w|)}$
  old_r: return $v_{(1,k-1)}rv_{(k,|v|)}aw_{(2,|w|)}$
  new_r: return $v_{(1,k-1)}rv_{(k,|v|)}aw_{(2,|w|)}$

The top-level algorithm of N is:

prepend the start state q_0 of M to the input;
while the input has not been exhausted,
  advance the computation by one M-step;
if the last non-blank symbol is a final state of M then
  accept
else
  reject

if q is predefined then
  for all transitions $d = (p' \xrightarrow{s/a} p)$ in the initial transition set of M,
    if $p' = q$ and $s = w_1$ then
      return $d, 0$;
  for each integer k from 1 to $|v|$,
    for all transitions $d = (p' \xrightarrow{s/a} p)$ added by action v_k,
      if $p' = q$ and $s = w_1$ then
        return d, k
else
  let r be the register of q;
  for each integer k from $|x| + 1$ to $|v|$,
    for all transitions $d = (p' \xrightarrow{s/a} p)$ added by action v_k,
      if $p' = old_r$ and $s = w_1$ then
        return d, k;
    if action v_k increments register r then
      break; (i.e. terminate the **for** loop)
  for each integer k from $|x|$ downto 1,
    for all transitions $d = (p' \xrightarrow{s/a} p)$ added by action v_k,
      if $p' = new_r$ and $s = w_1$ then
        return d, k;
    if action v_k increments register r then
      break; (i.e. terminate the **for** loop)
    for all transitions $d = (p' \xrightarrow{s/a} p)$ added by action v_k,
      if $p' = old_r$ and $s = w_1$ then
        return d, k;
reject (i.e. halt N in a nonaccepting state)

Figure 6: Algorithm to find an allowable transition

Prepending the start state to input w yields tape content $q_0 w$, which is the representation of the initial configuration of M on input w. Advancing the computation by one step can be accomplished via the previously given algorithms *without any additional storage*. If M reaches a dead end before exhausting the input, the M-step algorithm will reject. Otherwise, the top-level algorithm will accept iff M accepts. Hence $L(N) = L(M)$. Since N is deterministic and uses exactly $|w| + 1$ tape cells on input w, $L(N)$ is deterministic context-sensitive. □

Theorem 4.2 (SMFAs without λ-transitions)
If M is an SMFA without λ-transitions, then $L(M)$ is context-sensitive. □

Proof. The construction is just the same as for Theorem 4.1, except that, in the algorithm to find a transition, wherever the deterministic construction says to return values d, k, the modified N will nondeterministically choose to return, or continue searching for matching transitions. □

Theorem 4.3 (SMFAs with self-delete without λ-transitions)
If M is an SMFA with self-delete and without λ-transitions, then $L(M)$ is context-sensitive. Moreover if M is also deterministic, then $L(M)$ is deterministic context-sensitive. □

Proof. The construction is just the same as for Theorem 4.1, except for the specific alterations described below.

For each action a of M, let $T(a)$ be the set of all transitions added by a (expressed in ρ-relative form, of course).

Remove the actions of M from the tape alphabet, and add in their stead, for each action a of M, tape symbols $\sigma_{a,t}$ for every$t \subseteq T(a)$. The intent is that t indicates which of the transitions added by action a are still present.

When an action a is performed, instead of appending a to the string of actions on the tape, append the symbol $\sigma_{a,T(a)}$, indicating that a was performed and the added transitions are present.

In the algorithm to select an allowable transition, when iterating over all transitions added by an action a, skip over all the ones that have been deleted. When a transition is selected, if it was added by an action and is self-deleting, change the corresponding action symbol to indicate that that transition is no longer present.

Since there are only a finite number of predefined transitions, the finite state of N can be used to keep track of which of them are still present. When iterating over all predefined transitions, skip over all the ones that have been deleted; and if a self-deleting predefined transition is selected, remember that it is no longer present.

The resulting machine N accepts $L(M)$ in linear space, hence $L(M)$ is context-sensitive. Also, if M is deterministic, modifying N to always select the first allowable transition found does not alter the language accepted or the space complexity, but does render N deterministic, hence $L(M)$ is deterministic context-sensitive. $\square$

5 Finite-order SMFAs

Recall from [RS95a] that, *formally*, an SMA is defined as a 9-tuple, while *practically*, when the SMA is without deletion, the deletion function is usually omitted, leaving an 8-tuple.

In the special case of finite-order SMFAs, the action set and addition function can also be omitted as follows, leaving a 6-tuple.

Definition 5.1 (Normal form of actions)

Suppose M is an SMA, A the action set of M. For each $a \in A$, let $norm(a)$ be the closed-form standard representation of a (defined in [RS95a] Convention 3.7), if any. Then an action $a \in A$ is *in normal form* iff $a = norm(a)$.

M is *in normal form* iff all $a \in A$ are in normal form. $\square$

In a finite-order SMFA, every action has a standard representation in closed form; but in a general SMA (or even an unordered SMFA), this may not hold.

The standard representation of each action $a \in A$ encapsulates the behavior of the addition function of M on a, and on the actions of transitions directly or indirectly created by a. Therefore, if M is known to be in normal form, the effect of every action can be determined without further reference to the addition function. Moreover, the action set doesn't even have to be explicitly stated either, because the minimum set of actions needed can be deduced from the initial transition set.

These observations can also be extended to finite-order SMFAs with self-delete (see Convention 2.3), if a standard representation is provided for self-deleting actions.

Convention 5.2 (Notation)

The conventional representation of ρ-normal actions ([RS95a] Convention 3.7) is extended to actions of SMAs with self-delete, as follows.

Suppose M is an SMA with self-delete, and a a self-deleting action of M. If a has order zero in α, then a is represented by

$$SD$$

Otherwise, let $\underline{nd(a)}$ be the representation that would be given to the non-deleting variant of a. Then a is represented by

$$\underline{nd(a)}\,,\ SD$$

$\square$

For example, the following is the representation of a non-deleting action that adds a self-deleting action.

$$\text{add } new_r \xrightarrow{s/SD} old_r$$

Convention 5.3 (Finite-order SMFAs with self-delete)
Hereafter to the end of the paper, all finite-order SMFAs with self-delete are assumed to be in normal form unless explicitly stated otherwise; and when expressing a finite-order SMFA with self-delete as a tuple, the action set and addition function are usually omitted, so that the automaton is a 6-tuple $M = \langle \Sigma, Q_0, R, S, F, \delta_0 \rangle$ rather than an 8-tuple $M = \langle \Sigma, Q_0, R, A, S, F, \delta_0, \alpha \rangle$. □

In [RS95b], the tuple representation of SMFAs also omits the register set, leaving a 5-tuple. The explicit register set will be retained here, because in all but single-register machines, the registers are explicitly named in the normal-form actions.

Note that still another convention (5.5) is imposed at the end of §5.1.

5.1 Single-addition SMFAs

Theorem 5.4 (Single-addition finite-order SMFAs with self-delete)
Given any n^{th}-order m-register SMFA with self-delete M, a single-addition n^{th}-order $(2m+1)$-register SMFA with self-delete N can be constructed such that

- $L(N) = L(M)$.
- If M is without deletion, then N is without deletion.
- If M is deterministic, then N is deterministic up to trivial transitions.

□

Proof. Suppose $M = \langle \Sigma, Q_0, R, S, F, \delta_0 \rangle$ is a finite-order SMFA with self-delete. A finite-order SMFA with self-delete $N = \langle \Sigma, Q_N, R_N, S, F, \delta_N \rangle$ is constructed by successively altering M, as follows.

For each register $r \in R$, add a new register r'; and add one other new register r_x. The resulting set is $R_N = R \cup \{r' \mid r \in R\} \cup \{r_x\}$, and $|R_N| = 2|R| + 1$.

For each transition $d \in \delta_0$, add an infinite collection of predefined states $q_{d,k}$. Call the resulting set $Q'_0 = Q_0 \cup \{q_{d,k} \mid d \in \delta_0 \text{ and } k \in \mathbb{N}\}$. The predefined states of N will be a finite subset of these, $Q_N \in \mathcal{P}_\omega(Q'_0)$.

The heart of the construction of N is a function f that maps each action a of M to a nonempty string $f(a)$ of single-addition actions of N. Note that f is defined only on actions of M, hence a cannot involve states in $Q'_0 - Q_0$, or registers in $R_N - R$.

Suppose a has the form **add** $\{d_1, \cdots d_k\}$. If $k = 0$ then $f(a) = a$. Otherwise, the recursive algorithm of Figure 7 computes $f(a)$.

Let $l = \{\}$.
For each $(q \xrightarrow{s/b} p) \in \{d_1, \cdots d_k\}$,

let $q' = \begin{cases} old_{r'} & \text{if } q = new_r \\ q & \text{otherwise} \end{cases}$

let $p' = \begin{cases} old_{r'} & \text{if } p = new_r \\ p & \text{otherwise} \end{cases}$

append the following action to l:
add $q' \xrightarrow{s} new_{r_x}$
for each action b' in $f(b)$, from left to right,
append the following action to l:
add $old_{r_x} \xrightarrow{\lambda/b'} new_{r_x}$
append the following action to l:
add $old_{r_x} \xrightarrow{\lambda} p'$
For each $r \in R$,
if a increments r,
append the following actions to l:
add $old_{r'} \xrightarrow{\lambda} new_r$
add $old_r \xrightarrow{\lambda} old_{r'}$
add $new_{r'} \xrightarrow{\lambda} new_{r'}$
Let $f(a) = l$.

Figure 7: Algorithm to reduce an action to single-additions

Let $\delta_N = \{\}$.
For each $d = (q \xrightarrow{s/a} p) \in \delta_0$,
  let $a_1 \cdots a_k = f_a$, where the a_i are actions;
  add the following transition to δ_N:
    $q \xrightarrow{s} q_{d,0}$
  for i varying from 1 to k,
    add the following transition to δ_N:
      $q_{d,i-1} \xrightarrow{\lambda/a_i} q_{d,i}$
  add the following transition to δ_N:
    $q_{d,k} \xrightarrow{s} p$

Figure 8: Algorithm to reduce an initial transition set to single-action

On the other hand, suppose a has the form $SD(b)$, where b is of the form above (i.e. non-deleting), and SD maps any non-deleting action to its self-deleting variant. Let $b_1 \cdots b_k = f(b)$, where the b_i are actions. Then $f(SD(b)) = SD(b_1) \cdots SD(b_k)$.

The initial transition set of N is entirely disjoint from that of M. The algorithm to construct δ_N is shown in Figure 8.

The set Q_N of predefined states of N consists of exactly those elements of Q'_0 referenced in δ_N. Likewise, the set A_N of actions of N consists of exactly those actions referenced in δ_N (and the modification functions of N are chosen accordingly).

The SMFA N constructed as above satisfies the theorem. □

Convention 5.5 (single-addition)

Hereafter to the end of the paper, all SMFAs with self-delete are assumed single-addition except where explicitly otherwise stated. □

5.2 Ultralinear languages

In [RS95b], it was proved that all metalinear languages are accepted by single-register first-order SMFAs. This result will now be extended to the entire class of *ultralinear* languages. Basic definitions in the following treatment are adapted from [GS66].

Context-free grammars will be written in the form $G = \langle V, \Sigma, P, S \rangle$, where V is the set of nonterminals, Σ the set of terminals, P the set of productions, and $S \in V$ the start symbol of G.

Definition 5.6 (Linear language)

A context-free grammar $G = \langle V, \Sigma, P, S\rangle$ is *linear* iff each production in P is of the form $A \to uBv$ or $A \to u$, where $A, B \in V$ and $u, v \in \Sigma^*$.

A language is *linear* iff it is generated by some linear grammar. $\square$

Definition 5.7 (Metalinear language)

A context-free grammar $G = \langle V, \Sigma, P, S\rangle$ is *metalinear* iff each production in P is of the form $S \to \omega$, $A \to uBv$, or $A \to u$, where $\omega \in (\Sigma \cup V - \{S\})^*$, $A, B \in V - \{S\}$, and $u, v \in \Sigma^*$.

A language is *metalinear* iff it is generated by some metalinear grammar. $\square$

The metalinear languages are exactly the finite unions of concatenations of linear languages.

Definition 5.8 (Ultralinear language)

An *ultralinear decomposition* of a context-free grammar $G = \langle V, \Sigma, P, S\rangle$ is a finite list of sets $V_0, \cdots V_n$ that partitions V, such that each production in P is of the form $A_k \to uB_kv$ or $A_k \to \omega_k$, where $A_k, B_k \in V_k$, $u, v \in \Sigma^*$, and $\omega_k \in (\Sigma \cup \bigcup_{j<k} V_j)^*$.

A context-free grammar G is *ultralinear* iff there exists an ultralinear decomposition of G.

A language is *ultralinear* iff it is generated by some ultralinear grammar. $\square$

If G is linear, it has a one-set ultralinear decomposition $V_0 = V$. If G is metalinear, it has a two-set ultralinear decomposition $V_0 = V - \{S\}$, $V_1 = \{S\}$.

Theorem 5.9 (Ultralinear languages)

Every ultralinear language is accepted by some single-register first-order SMFA. $\square$

The following proof adapts and extends the proof in [RS95a] that every linear language is accepted by some first-order SMFA.

Proof. Suppose L is an ultralinear language over alphabet Σ.

Let $G = \langle V, \Sigma, P, S\rangle$ be an ultralinear grammar that generates L, and $V_0, \cdots V_n$ an ultralinear decomposition of G. Without loss of generality, assume that every production of G has the form $A_k \to uB_kv$, $A_k \to u$, or $A_k \to C_{k-1}D_{k-1}$, where $A_k, B_k \in V_k$, $u, v \in \Sigma_\lambda$, and $C_{k-1}, D_{k-1} \in V_{k-1}$. The assumption is without loss of generality because given any ultralinear grammar H, it is easy to construct an equivalent ultralinear grammar G and ultralinear decomposition $V_0, \cdots V_n$ of G of this form.

For each nonterminal $A \in V$, let $L(A)$ denote the set of terminal strings that can be derived from A using productions of P. Thus, $L = L(G) = L(S)$.

For every V_k and every $N \in V_k$, it will be shown (by induction on k) that an SMFA M to accept $L(N)$ can be constructed with all of the following properties.

- M is single-register and first-order.
- M has exactly one final state.
- In every reachable configuration of M,
  - The start state has in-degree zero.
  - The final state has out-degree zero.
  - State $\langle r, 0\rangle$ has degree zero.

Suppose $N \in V_k$, and for all $j < k$, if $B \in V_j$ then an SMFA with the above properties can be constructed to accept $L(B)$. Construct an SMFA $M = \langle \Sigma, Q_0, R, S, F, \delta_0\rangle$ to accept $L(N)$ as follows.

$R = \{r\}$. $F = \{q_f\}$.

Let $Q_0 = V_k\{S, q_x, q_f\}$. More predefined states may be added later.

Let $\delta_0 = \{\}$. More initial transitions will be added later.

Complete the machine as follows.

- Add an initial transition from q_0 to N labeled "λ/add $new \xrightarrow{\lambda} q_f$".
- For each production in P of the form $A_k \rightarrow uB_kv$, where $A_k, B_k \in V_k$ and $u, v \in \Sigma_\lambda$, add an initial transition from A_k to B_k labeled "u/add $new \xrightarrow{v} old$".
- For each production in P of the form $A_k \rightarrow u$, where $A_k \in V_k$ and $u \in \Sigma_\lambda$, add an initial transition from A_k to q_x labeled "u/add $q_x \xrightarrow{\lambda} old$".
- For each production in P of the form $A_k \rightarrow C_{k-1}D_{k-1}$, where $A_k \in V_k$ and $C_{k-1}, D_{k-1} \in V_{k-1}$,
  - Using predefined states not already in Q_0, construct an SMFA with the aforementioned properties accepting $L(C_{k-1})$; this is possible by inductive hypothesis. Add the predefined states and initial transitions of this machine to Q_0 and δ_0, respectively.
  - Using predefined states not already in Q_0, construct an SMFA with the aforementioned properties accepting $L(D_{k-1})$; this is possible by inductive hypothesis. Add the predefined states and initial transitions of this machine to Q_0 and δ_0, respectively.
  - Add an initial transition from A_k to the start state of the machine accepting $L(C_{k-1})$, labeled "λ/add $q_x \xrightarrow{\lambda} old$".
  - Add an initial transition from the final state of the machine accepting $L(C_{k-1})$ to the start state of the machine accepting $L(D_{k-1})$, labeled "λ".
  - Add an initial transition from the final state of the machine accepting $L(D_{k-1})$ to q_x, labeled "λ".

The SMFA so constructed accepts $L(N)$ and has all the requisite properties. Hence, by the principle of mathematical induction, the construction is possible for all $N \in V$. The machine constructed to accept $L(S)$ establishes the theorem. □

It is not known whether any single-register first-order SMFA can accept a context-free, non-ultralinear language. (Single-register first-order SMFAs to accept non-context-free languages are commonplace; see for example [RS95b].) However,

Theorem 5.10 (Non-ultralinear context-free languages)
There exists a first-order SMFA with two registers that accepts a non-ultralinear context-free language. □

Proof. This follows immediately from a construction in [RS93]. Figure 6 of that paper depicts a two-register first-order machine that accepts the set of all strings over $\{a, b\}$ with the same number of a's and b's; this language is known to be context-free but not ultralinear [Har78]. □

6 The pumping lemma

In [RS93, RS95b], it is conjectured that

Conjecture 6.1 (First-order register complexity classes)
For every integer $r \geq 1$, there exist languages with first-order register complexity r (that is, register complexity r in the class of all first-order SMFAs). □

If true, this would separate the first-order SMFA languages into an infinite hierarchy.

The immediate stimulus for this conjecture was languages of the form $\{w^{r+1} \mid w \in L\}$ for regular L and $r \geq 0$. Each such language is accepted by some first-order SMFA with r registers, and at the time it was conjectured that

Conjecture 6.2 (Complexity of w^{r+1})
There exist languages L such that for all $r \geq 0$, $\{w^{r+1} \mid w \in L\}$ has first-order register complexity r. □

Alas, Conjecture 6.2 turns out to be false (see §6.3); but it still appears likely that

Conjecture 6.3 (Complexity of $w^{f(r)}$)
There exists a non-decreasing function $f : \mathbb{N} \to \mathbb{N}$ and regular languages L such that $\{w^{f(r)} \mid w \in L\}$ has first-order register complexity r. □

Given the structure of these languages, it seems natural to look for some form of pumping lemma on r-register first-order SMFAs.

6.1 Priming the pump

Definition 6.4 (Computation)

Suppose $M = \langle \Sigma, Q_0, R, S, F, \delta_0 \rangle$ is an SMFA. A *computation* of M is a sequence of transitions that can be traversed by M starting from an initial configuration.

Suppose $c \in D^*$ is a computation of M, and $c = uvw$. Here, v is called a *substring instance* in c, to emphasize that its position in c is significant. Then $input(v)$ denotes the input string read by v, $source(v)$ denotes the state at which v begins, and $dest(v)$ denotes the state at which v ends. Note that because $c = uvw$, the source and destination of v are defined even when $v = \lambda$. Formally,

$$input(v) = \begin{cases} \lambda & \text{if } v = \lambda \\ st & \text{if } v = (p \xrightarrow{s/a} q)z \text{ and } t = input(z) \end{cases}$$

$$source(v) = \begin{cases} S & \text{if } u = \lambda \\ dest(u) & \text{otherwise} \end{cases}$$

$$dest(v) = \begin{cases} q & \text{if } v = z(p \xrightarrow{s/a} q) \\ source(v) & \text{otherwise} \end{cases}$$

Computation c is an *accepting* computation of M iff $dest(c) \in F$. Substring instance v of c is n^{th}-order iff all of the transitions in v are n^{th}-order. v is a *cycle in* c iff $|v| \geq 1$ and $source(v) = dest(v)$.

Substring instance v of $c = uvw$ is *productive in* c iff either $input(v) \neq \lambda$, or there exist x, y, z such that $c = uvxyz$, y is productive in c, and when M performs computation c, the transitions in y are created while traversing v. □

Note that for the productivity of a substring instance v of c, transitions created by v don't matter unless they are actually used by c. Also, a substring that occurs several times in c might be productive in some instances and non-productive in others.

Theorem 6.5 (Priming Lemma)

Suppose M is a finite-order SMFA, and $n \geq 0$. Then there exists $k \geq 0$ such that for every accepting computation c of M, if $|input(c)| \geq k$ then there must be some sequence of n consecutive productive cycles in c. □

That is, $c = uv_1v_2 \cdots v_nw$ such that all the v_i are productive cycles. Since the v_i are consecutive, $dest(v_i) = source(v_{i+1})$ for $1 \leq i < n$.

Proof. Suppose $M = \langle \Sigma, Q_0, R, S, F, \delta_0 \rangle$ is an m^{th}-order SMFA, and $n \geq 0$. Let $k = 1 + (n|Q_0| - 1) \sum_{0 \leq j \leq m} n^j$.

Suppose c is an accepting computation of M, and there is no sequence of n consecutive productive cycles in c. It will be shown that $|input(c)| < k$.

Let δ be the transition set of M after traversing the entire computation c. $\delta_0 \subseteq \delta$. For $1 \leq j \leq m$, let $\delta_j \subseteq \delta$ be the set of all transitions in δ that were created by instances of transitions in δ_{j-1}. Then $\delta_0, \delta_1, \cdots \delta_m$ is a partition of δ.

For $0 \leq j \leq m$, let $l_j \geq 0$ be the number of productive instances in c of transitions in δ_j. Then $|input(c)| \leq \sum_{0 \leq j \leq m} l_j$.

It will now be shown, by induction on j, that $l_j \leq (n|Q_0| - 1)n^j$. Therefore, $|input(c)| \leq \sum_{0 \leq j \leq m} l_j \leq (n|Q_0| - 1) \sum_{0 \leq j \leq m} n^j = k - 1$.

Base case: Suppose $j = 0$.

If any one predefined state $q \in Q_0$ were the source of more than n productive transition instances in c, then there would be a sequence of n consecutive productive cycles in c. Moreover, if $dest(c) \in Q_0$ were the source of more than $n - 1$ productive transition instances in c, then again there would be a sequence of n consecutive productive cycles in c. So the total number of productive transition instances in c with sources in Q_0 cannot exceed $n(|Q_0| - 1) + (n - 1) = (n|Q_0| - 1)$. But the source of an initial transition is always a predefined state; therefore, $l_0 \leq (n|Q_0| - 1)$.

Inductive step: Suppose $j > 0$, and $l_{j-1} \leq (n|Q_0| - 1)n^{j-1}$.

If a transition $d \in \delta_j$ has at least one productive instance in c, then by definition, the transition instance in c that created d is productive. Since $d \in \delta_j$, the transition that created it must be in δ_{j-1}; and there are only l_{j-1} productive instances of transitions in δ_{j-1}, each of which creates at most one transition in δ_j. Therefore, the number of transitions in δ_j that have productive instances in c cannot exceed l_{j-1}.

If there were more than n productive instances in c of any one transition in δ_j, then there would be a sequence of n consecutive productive cycles in c. So by the pigeon-hole principle, $l_j \leq nl_{j-1} \leq (n|Q_0| - 1)n^j$. $\Box$

6.2 Single-register

Theorem 6.6 (Single-register Pumping Lemma)

Suppose M is a single-register first-order SMFA. Then there exists an integer $k \geq 1$ such that for all $z \in L(M)$, if $|z| \geq k$ then there exist u, v, w, x, y with $z = uvwxy$, $|vx| \geq 1$, and for all $i \geq 1$, $uv^iwx^iy \in L(M)$. $\Box$

Proof. Suppose $M = \langle \Sigma, Q_0, \{r\}, S, F, \delta_0 \rangle$ is a single-register first-order SMFA, and $z \in L(M)$. Let $n = 2|Q_0|$, and let $k = 1 + (n|Q_0| - 1)(n + 1)$. Suppose $|z| \geq k$, and there do not exist u, v, w, x, y such that $z = uvwxy$, $|vx| \geq 1$, and for all $i \geq 1$, $uv^iwx^iy \in L(M)$. A contradiction will be derived.

Let $c \in D^*$ be the sequence of transitions traversed by M during some shortest accepting computation on z. Since $z = input(c)$ and $|z| \geq k$, by the Priming Lemma

(Theorem 6.5) there is a sequence of n consecutive productive cycles in c. Note that $|Q_0| \geq 1$, so $n \geq 2$.

Suppose $c = abd$ and b is a productive cycle. Let $u = input(a)$, $v = input(b)$, and $w = input(d)$. Hence $z = uvw$.

Suppose b is zeroth-order. Then since b is productive, $v \neq \lambda$. But since b does not modify the machine, $ab^i d$ is an accepting computation on input $uv^i w$ for all $i \geq 0$, contrary to assumption. So b must be strictly first-order.

Suppose none of the transitions created by c occur in d. Here again, since b is productive, $v \neq \lambda$. To help show that b can be pumped, the following additional notation is introduced.

For any sequence of transitions $\omega \in D^*$, let r_ω denote the change in the register value of M that would accrue from performing the actions on the transitions of ω. Thus in computation c, the register value just before M traverses b is r_a, and the value just after is r_{ab}. $r_{ab} - r_a = r_b$.

For any integers $i, j \geq 0$, let $t_{i,j} : Q \to Q$ and $\tau_{i,j} : D^* \to D^*$ be the following functions.

$$t_{i,j}(q) = \begin{cases} \langle r, h+j \rangle & \text{if } q = \langle r, h \rangle \text{ and } h > i \\ q & \text{otherwise} \end{cases}$$

$$\tau_{i,j}(\omega) = \begin{cases} \lambda & \text{if } \omega = \lambda \\ (t_{i,j}(q) \xrightarrow{s/a} t_{i,j}(p))\tau_{i,j}(\psi) & \text{if } \omega = (q \xrightarrow{s/a} p)\psi \end{cases}$$

Using this notation, $a\tau_{r_a,-r_b}(d)$ is an accepting computation of M on input uw. In fact, for all $i \geq 0$, $ab^i\tau_{r_a,r_b(i-1)}(d)$ is an accepting computation of M on input $uv^i w$, contrary to assumption. Therefore, d must contain at least one transition created by b.

Let $d = d_0 e_1 d_1 \cdots e_m d_m$ be the unique decomposition of d into substring instances d_i that contain no transitions created by b, and e_i that contain only transitions created by b, such that only d_0 and d_m are allowed to $= \lambda$. (That is, $e_1, \cdots e_m$ and $d_1, \cdots d_{m-1}$ have non-zero length.) So $c = abd_0 e_1 d_1 \cdots e_m d_m$. Because d contains at least one transition created by b, $m \geq 1$.

There is no way, in general, to construct an accepting computation from a and d without b, because there might be no way to get from d_{i-1} to d_i without the e_i created by b. However, with suitable transformations of d, b can be pumped $j \geq 1$ times, as follows.

Transitions created by b can only begin and end on states $q \in Q_0$, or $\langle r, h \rangle$ for $r_a \leq h \leq r_{ab}$. Transitions *not* created by b cannot begin or end on states $\langle r, h \rangle$ for $r_a < h < r_{ab}$. Therefore, any endpoint of an e_i that is also an endpoint of a $d_{i'} \neq \lambda$ must belong to the set $Q_0 \cup \{\langle r, r_a \rangle, \langle r, r_{ab} \rangle\}$. If $d_m = \lambda$, then e_m ends on a state $q \in F \subseteq Q_0$. If $d_0 = \lambda$, then since b is a cycle, e_1 begins on a state q that existed before b, hence $q \in Q_0 \cup \{\langle r, r_a \rangle\}$.

Thus, all e_i begin and end on states in the set $Q_0 \cup \{\langle r, r_a\rangle, \langle r, r_{ab}\rangle\}$.

Suppose $j \geq 2$. An accepting computation c' of M will now be constructed by pumping b j times and transforming each d_i and e_i as follows.

$$c' = ab^j d'_0 e'_1 d'_1 \cdots e'_m d'_m$$

The d_i are transformed as in the earlier construction for $m = 0$: $d'_i = \tau_{r_a, r_b(j-1)}(d_i)$.

For each e_i, an e'_i must be constructed that will connect $dest(d'_{i-1})$ to $source(d'_i)$. The tools used in this construction will be the transition paths analogous to e_i created by the j iterations of b in c'. For $1 \leq h \leq j$, the h^{th} iteration of b creates a transition path $\tau_{r_a, r_b(h-1)}(e_i)$; note in particular that the first iteration of b creates path $\tau_{r_a,0}(e_i) = e_i$.

If neither endpoint of e_i is $\langle r, r_{ab}\rangle$, then the endpoints of e'_i are the same as those of e_i; so let $e'_i = e_i$. Otherwise, if neither endpoint of e_i is $\langle r, r_a\rangle$, let $e'_i = \tau_{r_a, r_b(j-1)}(e_i)$.

Otherwise, e_i must go either from $\langle r, r_a\rangle$ to $\langle r, r_{ab}\rangle$ or vice versa, and $r_b \neq 0$. If $source(e_i) = \langle r, r_a\rangle$, let $e'_i = \tau_{r_a,0}(e_i)\tau_{r_a,r_b}(e_i) \cdots \tau_{r_a, r_b(j-1)}(e_i)$; otherwise, let $e'_i = \tau_{r_a,r_b(j-1)}(e_i)\tau_{r_a,r_b(j-1)}(e_i) \cdots \tau_{r_a,0}(e_i)$.

This completes the construction of accepting computation c'.

For each d_i, $input(d'_i) = input(d_i)$; and for each e_i, either $input(e'_i) = input(e_i)$ or $input(e'_i) = input(e_i)^j$. So there exist unique substring instances $w_0, \cdots w_l$ in z, and unique non-λ substring instances $x_1, \cdots x_l$ in z, such that

$$\begin{aligned} z &= w_0 x_1 w_1 \cdots x_l w_l \\ input(c') &= w_0 x_1^j w_1 \cdots x_l^j w_l \end{aligned}$$

Note that when $v \neq \lambda$, $w_0 = u$ and $x_1 = v$. For each x_i with $i \geq 2$ or $v = \lambda$, there is an $e_{i'}$ such that $x_i = input(e_{i'})$, and the endpoints of $e_{i'}$ are $\langle r, r_a\rangle$ and $\langle r, r_{ab}\rangle$.

Suppose b is the rightmost cycle for which $l \geq 1$. (The case that no such b exists will be considered momentarily.) By assumption, there are no one or two pumpable substring instances in z, so $l \geq 3$. Hence there must be integers $i \neq i'$ such that $e_i, e_{i'}$ each accepts non-λ input, and has endpoints $\langle r, r_a\rangle$ and $\langle r, r_{ab}\rangle$. But then, at least one of these states must be the endpoint of a cycle to the right of b that accepts non-λ input, contrary to our choice of b. So $l = 0$ for all cycles b.

Since b is productive but $v = \lambda$, there must be some e_i such that $input(e_i) \neq \lambda$. e_i cannot be a cycle (because it accepts non-λ input). e_i cannot have endpoints $\langle r, r_a\rangle$ and $\langle r, r_{ab}\rangle$, because that would cause $l \geq 1$. So e_i must begin or end on a predefined state.

Recall that because $|z| \geq k$, there exists a sequence of $n = 2|Q_0|$ consecutive productive cycles in c. For each such cycle b_i in c, there is a substring instance f_i in c that accepts non-λ input, and either begins or ends on a predefined state; and no two of the f_i overlap. Because there are at least $2|Q_0|$ such f_i, by the pigeon-hole principle there must either be some f_i that begins on the final state of c, or some two

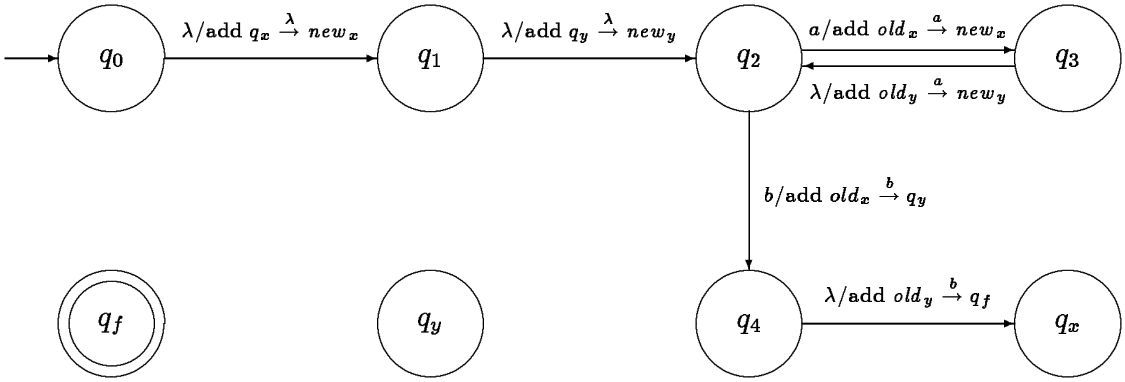


Figure 9: 2-register SMFA to accept $\{www \mid w = a^n b \text{ for some } n \geq 0\}$

f_i that begin on the same state, or some two f_i that end on the same state. In any of these cases, there will be a cycle in c accepting non-λ input.

Since a contradiction has been reached, there must exist u, v, w, x, y such that $z = uvwxy$, $|vx| \geq 1$, and for all $i \geq 1$, $uv^iwx^iy \in L(M)$. □

Corollary 6.7 (Examples)
The following languages have first-order register complexity 2.

$$\{www \mid w = a^n b \text{ for some } n \geq 0\}$$

$$\{a^n b^n c^n \mid n \geq 0\}$$

□

Proof. That these languages cannot be accepted with just one register follows easily from the Single-register Pumping Lemma (Theorem 6.6). 2-register machines to accept them are shown in Figures 9 and 10. □

Theorem 6.6 is rather weak. It says nothing about the positions within z of the substring instances to be pumped, and puts no upper bound on their length. Consequently, it has nothing to say about some important languages. For example, $L = \{www \mid w \in \Sigma^*\}$ satisfies the condition of the theorem, because for every $z \in L$ and $i \geq 0$, $z^i \in L$. Moreover, it is particularly unfortunate that *every* context-free language satisfies the condition of the theorem.

6.3 Multi-register

An important property of single-register first-order SMFAs is that *any cycle in an accepting computation can be pumped* in such a way that a new accepting computation

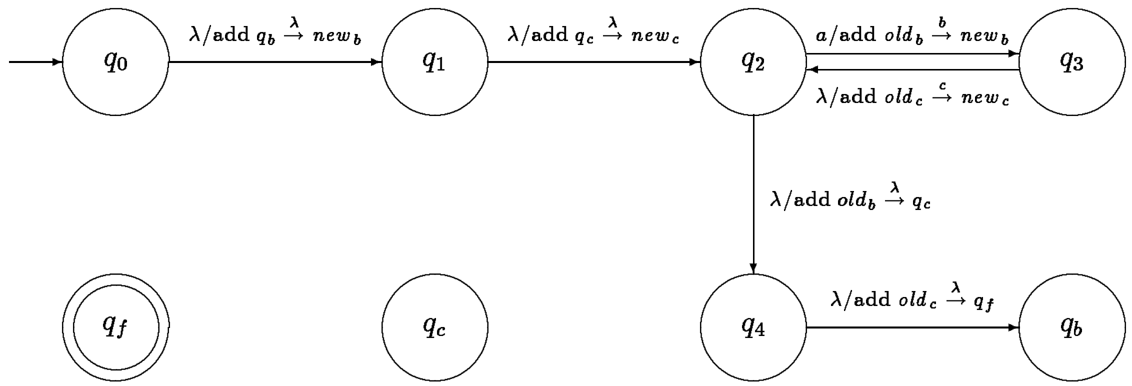


Figure 10: 2-register SMFA to accept $\{a^nb^nc^n \mid n \geq 0\}$

can still be constructed. In the proof of the Single-register Pumping Lemma (Theorem 6.6), this property was used to show that no more than 2 substring instances of the input must be pumped.

Unfortunately, this property does not hold for first-order SMFAs with more than one register. Figure 11 illustrates the problem. Here, the predefined path from q_z back to q_x must be traversed *exactly* once in every accepting computation; if it is traversed more than once, or not at all, the machine cannot possibly accept.

Figure 11 is a modification of Figure 9. Just as the latter figure could easily be modified to accept $\{w^3 \mid w \in L\}$ for any regular L, the former could be modified to accept $\{w^5 \mid w \in L\}$ for any regular L, thus disproving Conjecture 6.2. A slight modification of the machine, with the cycle on q_y instead of q_z, would accept $\{w^4 \mid w \in L\}$ for regular L.

Although this modification technique provides 2-register languages in which 4 or 5 substring instances have to be pumped simultaneously, there are still only 3 distinct substrings involved; it's just that two of these substrings are allowed to have two instances each. It still appears likely that no 2-register machine can accept $\{a^nb^nc^nd^ne^n \mid n \geq 0\}$.

7 Conclusion

This paper has presented new results on the computational power of various classes of SMFAs (Self-Modifying Finite Automata), notably on the class of single-register first-order SMFAs. These results extend and augment results previously presented in [RS93, RS94, RS95a, RS95b]. The treatment of number of SMFA registers as a complexity measure was proposed. The property of path determinism was formally defined, and investigated in relation to both computational power and register complexity.

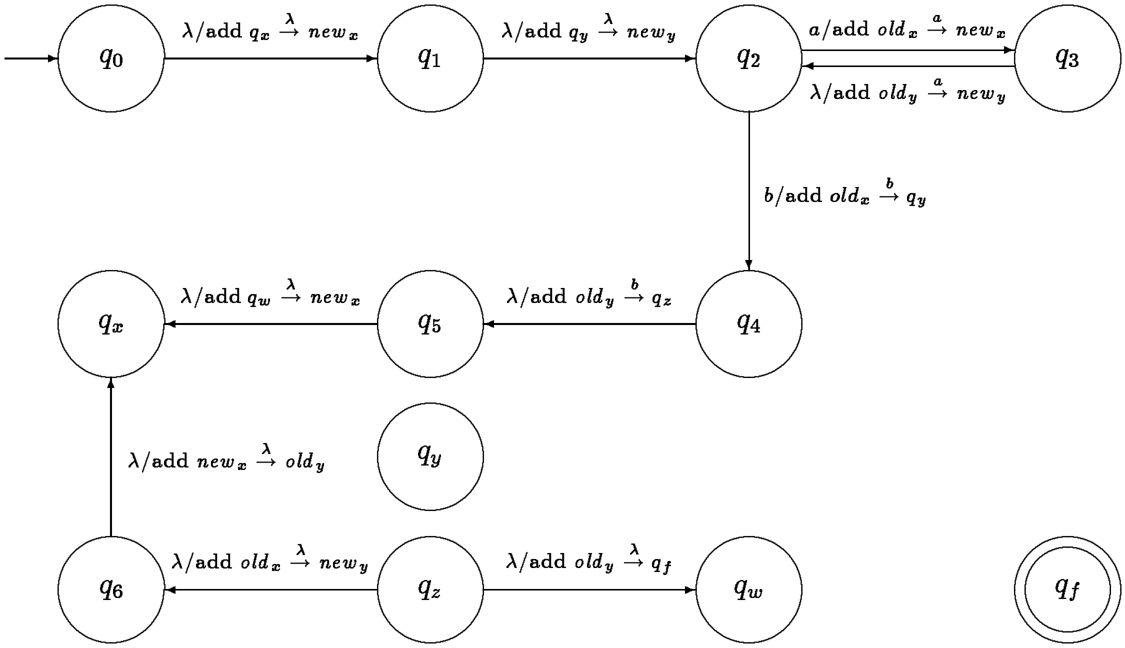


Figure 11: 2-register SMFA to accept $\{w^5 \mid w = a^n b \text{ for some } n \geq 0\}$

An important outstanding conjecture on SMFAs was resolved. First proposed in [RS93], the conjecture states that the first-order register complexity of $\{w^{r+1} \mid w \in L\}$ is r for some regular L. The conjecture was shown to be true for $r \leq 2$, but false for $r \geq 3$. A weaker variation on the conjecture, consistent with proven results to date, was proposed.

These conjectures are closely allied to the problem of formulating a pumping lemma for n-register first-order SMFAs. A weak single-register pumping lemma was derived, but the n-register case remains outstanding. Further research is indicated, both to develop a lemma for n-register machines, and to provide a stronger single-register lemma.

References

[GS66] S. Ginsburg and E. H. Spanier. Finite-turn pushdown automata. *SIAM J. Control*, 4(3):429–453, Aug. 1966.

[Har78] M. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Massachusetts, 1978.

[RS93] R. Rubinstein and J. Shutt. Self-modifying finite automata. Technical Report WPI-CS-TR-93-11, Worcester Polytechnic Institute, Worcester, MA, December 1993.

[RS94] R. Rubinstein and J. Shutt. Self-modifying finite automata. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94 Vol. I: Proc. 13th IFIP World Computer Congress*, pages 493–498, Amsterdam, 1994. North-Holland.

[RS95a] R. Rubinstein and J. Shutt. Self-modifying finite automata — basic definitions and results. Technical Report WPI-CS-TR-95-2, Worcester Polytechnic Institute, Worcester, MA, August 1995.

[RS95b] R. Rubinstein and J. Shutt. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4):185–190, November 1995.