JOURNAL OF COMPUTER AND SYSTEM SCIENCES 14, 344–359 (1977)

Equivalence Problems for Deterministic Context-Free Languages and Monadic Recursion Schemes

EMILY P. FRIEDMAN*

Computer Science Department, University of California, Los Angeles, California 90024

Received September 29, 1975; revised September 30, 1976

The equivalence problem for deterministic context-free languages is shown to be decidable if and only if the strong equivalence problem for monadic recursion schemes is decidable.

1. INTRODUCTION

In this paper we consider how certain properties of monadic recursion schemes are related to those of deterministic context-free languages. Previous work showed only how to translate properties of schemes into properties of languages [3]. This was accomplished by a construction that when given any monadic recursion scheme S, produced a deterministic pushdown automaton (abbreviated dpda) accepting a language encoding the free interpretations and associated values of S (interpreted value language of S [3]). Since two monadic recursion schemes are strongly equivalent if and only if their interpreted value languages are equal, the strong equivalence problem for monadic recursion schemes is reducible to the equivalence problem for deterministic context-free languages [3]. For the first time we are able to prove the validity of the converse of this result. That is, given any two dpda's, we can construct two monadic recursion schemes such that the schemes are strongly equivalent if and only if the languages accepted by the dpda's are equal.

Section 2 contains definitions and terminology about devices known as jump pushdown automata. Such a machine is essentially a dpda that can erase the pushdown store down to and including some specified symbol with only one move. Known results about these devices are used to show that any deterministic context-free language can be accepted by a dpda that changes state on consecutive e-moves only a bounded number of times. This is an important tool for proving results in Section 3.

Section 3 presents a family of pushdown acceptors referred to as extended simple

* The results reported are a portion of the author's Ph.D. dissertation written at Harvard University under the supervision of Professor Ronald V. Book. They were presented in a different form at the IEEE Fifteenth Annual Symposium on Switching and Automata Theory, October 1974. The research was supported in part by the National Science Foundation under Grant NSF GJ-30409 and Grant NSF GJ-803.

ISSN 0022-0000

machines. An extended simple machine is a single-state dpda that has the ability to keep its read head situated over the same input symbol for more than one move. We show that any language accepted by such a device must be a prefix-free deterministic context-free language. On the other hand, we demonstrate that not every prefix-free deterministic language can be accepted by an extended simple machine. Nevertheless, using the results of Section 2, we prove that the equivalence problem for dpda's is reducible to the equivalence problem for extended simple machines.

In Section 4 we define monadic recursion schemes and establish translations between them and extended simple machines in both directions. First, given any monadic recursion scheme S, we show how to construct an extended simple machine that accepts the interpreted value language of S. Then, we provide a construction so that given an extended simple machine M, we can find a scheme with interpreted value language that "encodes" the operation of M. Combining these translatability results with those of Section 3 give us the main result of the paper: The equivalence problem for deterministic pushdown automata is decidable if and only if the strong equivalence problem for monadic recursion schemes is decidable.

2. DPDA's with Jumps

In this section we investigate deterministic pushdown automata augmented with jump instructions [5]. Such a device is essentially a dpda with the added capability to erase (in one move) the pushdown store down to and including the topmost occurrence of some specified pushdown symbol. We use these machines as an intermediate tool for proving that the deterministic context-free languages can be characterized by deterministic pushdown automata (without jump instructions) that change state on consecutive e-moves at most a bounded number of times.

2.1. Definition. A *jump deterministic pushdown automaton* (abbreviated jdpda) $M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ consists of a finite set K of states, a set $F \subseteq K$ of final states, a finite input alphabet Σ, a finite pushdown alphabet Γ, an initial state $q_0 \in K$, an initial pushdown symbol $Z_0 \in \Gamma$, and a (partial) transition function δ: $K \times (\Sigma \cup \{e\}) \times \Gamma \to (K \times \Gamma^*) \cup (\{J\} \times K \times \Gamma)$. For each combination of state q and pushdown symbol Z, we require that either $\delta(q, e, Z) = \varnothing$ (i.e., undefined) or $\delta(q, a, Z) = \varnothing$ for each $a \in \Sigma$. A transition of the form $\delta(p, a, Z) = (J, q, Y)$ is called a jump instruction, and it causes the pushdown store to be erased down to and including the topmost occurrence of symbol Y. If no Y occurs anywhere in the store, then no move is possible, and the machine halts.

A configuration of M is described by (q, w, α), where q is the current state, $w \in \Sigma^*$ is portion of the input tape remaining to be read, and $\alpha \in \Gamma^*$ is the current contents of the pushdown store, with the topmost symbol of the store at the left of α. Moves of the jdpda are described by the relation $\underset{M}{\vdash}$ on configurations in the usual way [4, 5, 6]. We let $\overset{*}{\underset{M}{\vdash}}$ denote the transitive reflexive closure of $\underset{M}{\vdash}$, and $\overset{+}{\underset{M}{\vdash}}$ the transitive closure. For $t \geq 0$, the operation $\overset{t}{\underset{M}{\vdash}}$ denotes a sequence of t consecutive moves as determined by $\underset{M}{\vdash}$.

If there is a $d \geqslant 0$ such that $(p, u, \alpha) \vdash_M^t (q, u, \beta)$ implies $t \leqslant d$, then M operates with finite delay and delay d.

Acceptance of an input tape is defined in three ways:

(i) empty store: $N(M) = \{w \mid (q_0, w, Z_0) \vdash_M^* (q, e, e) \text{ for some } q \in K\}$,

(ii) final state: $T(M) = \{w \mid (q_0, w, Z_0) \vdash_M^* (q, e, \alpha) \text{ for some } q \in F, \alpha \in \Gamma^*\}$,

(iii) simultaneous final state and empty store:

$$L(M) = \{w \mid (q_0, w, Z_0) \vdash_M^* (q, e, e), q \in F\}.$$

Except for the jump instructions, a jdpda operates exactly like the standard definition for a dpda [4, 5]. Moreover, these jump instructions do not enlarge the family of languages accepted, since they can be simulated in the obvious way with e-moves. Henceforth, when we write dpda, we imply a dpda without jumps.

The family of languages accepted by a dpda on final state, denoted D, are the *deterministic context-free languages.* Those accepted either by empty store alone or by simultaneous empty store and final state are the *prefix-free deterministic context-free languages,* and are denoted P. Here, prefix-free means that no proper prefix of a word in the language can also be in the language.

One of the long-standing open problems in language theory is whether or not there is a decision procedure for determining if two dpda's accept (by final state) the same language. This question is called the *equivalence problem for deterministic context-free languages*, and its solution is thought to have far-reaching effects. In this paper, we consider how this problem is related to one in the area of schema theory.

For technical purposes, we wish to show that every language in P is accepted by some dpda that changes state on consecutive e-moves at most a bounded number of times. We do not claim the existence of a bound on the number of consecutive e-moves, but rather a bound on how many times the machine changes state during these e-moves. This will be a useful intermediate step in obtaining the results in the next section. For this, we rely on a result proven in [5].

2.2. LEMMA (Greibach). *For any* dpda M_1, *one can effectively construct a* jdpda M_2 *that operates with finite delay such that* $N(M_2) = N(M_1)$.

Thus, every language in P is accepted by some jdpda that operates with finite delay. This is contrasted to the well-known fact that there exist languages in P that cannot be accepted by a dpda that operates with finite delay [4, 8].

We now consider the result alluded to above.

2.3. THEOREM. *Let M be any* dpda. *Then there exists a* jdpda $M'' = (K'', \Sigma'', \Gamma'', \delta'', q_0'', Z_0'', F'')$ *with constant* $c \geqslant 0$ *such that* $N(M) = N(M'')$ *and if*

$$(p_0, v, \alpha_0) \vdash_{M''}^* (p_1, v, \alpha_1) \vdash_{M''}^* \cdots \vdash_{M''}^* (p_t, v, \alpha_t)$$

for

$$p_0, p_1, \ldots, p_t \in K'', \quad v \in (\Sigma'')^*, \quad \alpha_0, \alpha_1, \ldots, \alpha_t \in (\Gamma'')^*,$$

where

$$p_i \neq p_{i+1}, \qquad 0 \leqslant i \leqslant t-1,$$

then

$$t \leqslant c.$$

Proof. Let M be any dpda. Then by Lemma 2.2 we know that there exists some jdpda $\bar{M} = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ that operates with delay d for some $d \geqslant 0$, and $N(M) = N(\bar{M})$.

We construct a dpda M' (without jumps) from the jdpda $\bar{M}$ that simulates the operation of $\bar{M}$ in the obvious manner: Jump instructions in $\bar{M}$ are simulated by consecutive e-moves of M' that pop the store until the desired symbol is found. At most two state changes are needed to accomplish this simulation. Since a nonjump instruction in $\bar{M}$ is the same in M', and a jump e-move of $\bar{M}$ causes at most two changes of state to simulate in M', we find that a computation in $\bar{M}$ that consists of m e-moves is simulated in M' by a computation of e-moves that change state at most $2 \times m$ times. Of course, this computation in M' may be of length much greater than $2 \times m$; we are only claiming that there are less than $2 \times m$ changes of state.

Formally, we construct $M' = (K', \Sigma, \Gamma, \delta', q_0, Z_0, K)$ as follows. Let $K' = K \cup (K \times \Gamma)$, and define δ' to be: For $p, q \in K$, $a \in \Sigma \cup \{e\}$, $\alpha \in \Gamma^*$, $Z_i, Z_j, Z_k \in \Gamma$,

(a) $\delta'(p, a, Z_k) = \delta(p, a, Z_k)$ iff $\delta(p, a, Z_k) = (q, \alpha)$,

(b) $\delta'(p, a, Z_k) = ([q, Z_i], Z_k)$ iff $\delta(p, a, Z_k) = (J, q, Z_i)$,

(c) $\delta'([q, Z_i], e, Z_i) = (q, e)$,

(d) $\delta'([q, Z_i], e, Z_j) = ([q, Z_i], e)$, if $i \neq j$.

The state $[q, Z_i]$, for $q \in K$, $Z_i \in \Gamma$, is used to encode the fact that machine M' is simulating a jump instruction of $\bar{M}$ that erases down through the topmost occurrence of Z_i and then changes the state of $\bar{M}$ to q. In M', $[q, Z_i]$ indicates that consecutive e-moves must pop the pushdown store, keeping the state of M' to be $[q, Z_i]$, until *a* Z_i is encountered on the store. At this point, an e-move pops the Z_i and changes the state of M' to q. Since K is the set of final states of M', it should be clear that $L(M') = N(\bar{M})$. Using the standard construction [6], we can find a dpda M'' with $N(M'') = L(M') = N(\bar{M})$. ▌

3. Extended Simple Languages

In this section we define a family of pushdown store acceptors known as extended simple machines. An extended simple machine is a single-state dpda with an added feature that allows its read head to remain situated over the same input symbol for more than one move. We show that the family of languages accepted by the extended simple machines is properly contained in P. Then, we prove that it is decidable whether two dpda's accept the same language if and only if it is decidable whether two extended simple machines accept the same language.

A dpda is said to be simple if it has only a single state [7]. Because the equivalence problem for languages accepted by simple dpda's is known to be decidable [7], it would be interesting to find that the equivalence problem for dpda's (not necessarily simple) can be reduced to this one; this problem remains open. We introduce a new feature into the definition of simple machines as a means of extending the family of languages accepted. The equivalence problem for this new family of languages will be shown to be equivalent to the equivalence problem for P. The feature that we consider is that of giving the read head the option on each move of advancing one symbol to the right or of remaining situated over the current tape symbol. Thus, such a device can make an unbounded number of moves without advancing the read head. One immediate consequence is that we can pop the entire pushdown store when some distinguished tape symbol is encountered. We refer to this component as the delay feature, and we will call simple machines with this new delay feature extended simple machines. Because these devices possess only one state, no information can be stored in the finite-state control. Therefore, they can alternately be thought of as stateless devices, so there is no necessity for including a state set in the definition. Instead, we can simply use notation and use the definition that follows.

3.1. Definition. An *extended simple machine* (abbreviated es machine) is a four-tuple $M = (\Sigma, \Gamma, \delta, Z_0)$, where Σ is a finite set of input symbols, Γ is a finite set of pushdown symbols, $Z_0 \in \Gamma$ is the initial pushdown symbol, and δ is a partial transition function δ: $\Sigma \times \Gamma \to \Gamma^* \times \{0, 1\}$. A configuration is a pair (w, α), where $w \in \Sigma^*$ is the portion of the portion of the input remaining to be read, and $\alpha \in \Gamma^*$ is the current contents of the pushdown store, with the topmost symbol of the store at the left of α. We define the operation $\vdash_{\overline{M}}$ to indicate a move on configurations of M as follows: For $a \in \Sigma$, $w \in \Sigma^*$, $A \in \Gamma$, $\alpha, \beta \in \Gamma^*$,

$$\text{(i)} \qquad (aw, A\alpha) \vdash_{\overline{M}} (w, \beta\alpha) \quad \text{iff} \quad \delta(a, A) = (\beta, 1),$$

$$\text{(ii)} \qquad (aw, A\alpha) \vdash_{\overline{M}} (aw, \beta\alpha) \quad \text{iff} \quad \delta(a, A) = (\beta, 0).$$

Define $\overset{*}{\vdash}_{\overline{M}}$ to be the transitive reflexive closure of $\vdash_{\overline{M}}$, and $\overset{+}{\vdash}_{\overline{M}}$ to be the transitive closure of $\vdash_{\overline{M}}$. For $t \geqslant 0$, we define $\overset{t}{\vdash}_{\overline{M}}$: To be a sequence of t consecutive moves as determined by $\vdash_{\overline{M}}$.

The language accepted by M, denoted $N(M)$, is called an *extended simple language*, where $N(M) = \{w \in \Sigma^* \mid (w, Z_0) \overset{*}{\vdash}_{\overline{M}} (e, e)\}$.

Each occurrence of $c_1 \vdash_{\overline{M}} c_2$ is called a move of the es machine M. Moves of the form (i) above advance the input one tape symbol, whereas moves of the form (ii) do not advance the input. The latter moves are somewhat akin to the e-moves of a dpda. In fact, if we were allowed multiple states in the finite-state control, these non-advancing moves could be simulated by e-moves. This is exemplified in the proof of the following theorem.

3.2. THEOREM. *For any* es *machine* $M = (\Sigma, \Gamma, \delta, Z_0)$, *one can effectively construct a* dpda M' *such that* $L(M') = N(M)$.

Proof. Construct dpda $M' = (K, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_0\})$, where $K = \{q_0\} \cup \{P_\sigma \mid \sigma \in \Sigma\}$, and for all $a \in \Sigma$, $Z \in \Gamma$, $\alpha \in \Gamma^*$,

$$\begin{aligned}
\delta'(q_0, a, Z) &= (q_0, \alpha), &&\text{if } \delta(a, Z) = (\alpha, 1),\\
&= (P_a, \alpha), &&\text{if } \delta(a, Z) = (\alpha, 0),\\
&= \varnothing, &&\text{if } \delta(a, Z) = \varnothing,\\[1em]
\delta'(P_a, e, Z) &= (q_0, v), &&\text{if } \delta(a, Z) = (v, 1),\\
&= (P_a, v), &&\text{if } \delta(a, Z) = (v, 0),\\
&= \varnothing, &&\text{if } \delta(a, Z) = \varnothing.
\end{aligned}$$

We make the following claim about the behavior of M'. For all $w \in \Sigma^*$, $(w, Z_0) \overset{*}{\underset{M}{\vdash}} (e, e)$ iff $(q_0, w, Z_0) \overset{*}{\underset{M'}{\vdash}} (q_0, e, e)$.

The proof of this claim is a straightforward induction on the length of a derivation, and is omitted. It follows that $L(M') = N(M)$. ▌

From this result, we get the following corollary.

3.3. COROLLARY. *The decidability of the equivalence problem for languages in P implies the decidability of the equivalence problem for* es *languages.*

Although this result is not too surprising in itself, the validity of its converse may be. Recall that Theorem 3.2 shows that any language accepted by an es machine can also be accepted by a dpda. The converse to this result is not true. To illustrate this, consider the language $L = \{a^nba^nb \mid n \geqslant 1\} \cup \{a^nca^nc \mid n \geqslant 1\}$. L is certainly accepted by a dpda, but L is not accepted by any es machine. An es machine has no mechanism for "remembering" whether it reads b or c before processing the second segment of a's. In light of this, the next result may seem somewhat surprising.

3.4. THEOREM. *The equivalence problem for languages in P is reducible to the equivalence problem for* es *languages.*

Proof. Given a dpda M, we construct an es machine E_M that accepts a homomorphic image of $N(M)$. This new language looks like $N(M)$, but each string in $N(E_M)$ has a fixed, distinguishable string of symbols inserted between every two symbols of the corresponding string in $N(M)$. These inserted strings play the role of "markers," allowing the es machine to encode the states of M by means of its delay feature.

Let $M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a dpda. Without loss of generality, assume that $K = \{q_0, ..., q_{n-1}\}$ for some $n \geqslant 1$, and that there exists some integer $b \geqslant 0$ such that M changes state at most b times during consecutive e-moves.

Let $S_M = \{\#_i^d \mid 0 \leqslant d \leqslant b,\ 0 \leqslant i \leqslant n-1\}$ be a set of symbols disjoint from Σ and let w_M be the string

$$\#_0^0\#_1^0 \cdots \#_{n-1}^0\#_0^1\#_1^1 \cdots \#_{n-1}^1 \cdots \#_0^b\#_1^b \cdots \#_{n-1}^b .$$

We define a homomorphism $h_M : \Sigma^* \to (\Sigma \cup S_M)^*$ determined by defining $h_M(a) = a \cdot w_M$.

We shall construct an es machine E_M such that $N(E_M) = w_M \cdot h_M(N(M))$. Thus, the language accepted by es machine E_M looks like $N(M)$: Each string in $N(E_M)$ has the string w_M inserted between every two symbols and attached to both ends of corresponding string in $N(M)$. For example, suppose that the tape $ac \in N(M)$. Then, if we consider the case where $n = 2$ (3 states), $b = 1$, we have

$$\underbrace{\#_0^0\#_1^0\#_2^0\#_0^1\#_1^1\#_2^1}_{w_M}\ \underset{a}{a}\ \underbrace{\#_0^0\#_1^0\#_2^0\#_0^1\#_1^1\#_2^1}_{w_M}\ \underset{c}{c}\ \underbrace{\#_0^0\#_1^0\#_2^0\#_0^1\#_1^1\#_2^1}_{w_M} \in N(E_M).$$

For each integer i, $0 \leqslant i \leqslant n-1$, the symbols in $\{\#_i^d \mid 0 \leqslant d \leqslant b\}$ will be interpreted as "encodings" of state q_i. Notice that since the number of elements in the set is $b+1$, there are precisely $b+1$ encodings of state q_i. We shall see that if a computation in E_M gets to a point where it is reading an input symbol like $\#_i^d$ and the topmost pushdown symbol is $A \in \Gamma$, then E_M is considered to be "simulating" state q_i with A on the top of M's pushdown store. The superscript d is essential for the simulation, as will become apparent later.

Now we formally define es machine E_M. Construct $E_M = (\Sigma \cup S_M, \Gamma_M, \delta_M, \bar{Z}_0)$, where

$$\begin{aligned} \Gamma_M = \Gamma &\cup S_M \cup \{\bar{Z}_0, B\} \\ &\cup \{Q_i^d \mid 0 \leqslant d \leqslant b, 0 \leqslant i \leqslant n-1\}, \\ &\cup \{[q_i, A] \mid q_i \in K, A \in \Gamma\}, \end{aligned}$$

$$(\Gamma \cap S_M \cap \{\bar{Z}_0, B\} \cap \{Q_i^d\} \cap \{[q_i, A]\} = \varnothing)$$

and the transition function δ_M is defined as follows:

For every

$$\begin{array}{ll} A \in \Gamma, & \alpha \in \Gamma^*, \\ q_i, & q_j \in K, \\ a \in \Sigma, & u \in \Sigma^*, \\ d, & 0 \leqslant d \leqslant b, \\ k, & 0 \leqslant k \leqslant n-1. \end{array}$$

I. *Initialization: Introduce B as bottom-of-store indicator*

(a) $$\delta_M(\#_0^0, \bar{Z}_0) = (Z_0B, 0).$$

II. *Simulation of an e-move that does not change state*

(b) If

$$\delta(q_i, e, A) = (q_i, \alpha),$$

then

$$\delta_M(\#_i^d, A) = (\alpha, 0).$$

III. *Simulation of an e-move that causes a change of state*

(c) If

$$\delta(q_i, e, A) = (q_j, \alpha), \qquad q_i \neq q_j,$$

then for $d < b$

$$\delta_M(\#_i^d, A) = (\#_{i+1}^d \cdots \#_{n-1}^d \#_0^{d+1} \cdots \#_{j-1}^{d+1} Q_j^{d+1} \alpha, 1).$$

Notation. If $i = n - 1$, then $\#_{i+1}^d \cdots \#_{n-1}^d = e$, if $j = 0$, then $\#_0^{d+1} \cdots \#_{j-1}^{d+1} = e$,

(d) $$\delta_M(\#_k^d, \#_k^d) = (e, 1),$$

(e) $$\delta_M(\#_k^d, Q_k^d) = (e, 0).$$

IV. *Simulation of a move that is not an e-move*

If $\delta(q_i, a, A) = (q_j, \alpha)$, then

(f) $$\delta_M(\#_i^d, A) = (\#_{i+1}^d \cdots \#_{n-1}^b [q_i, A], 1).$$

Notation. $\#_{i+1}^d \cdots \#_{n-1}^b$ denotes the (possibly empty) portion of string w_M to the right of symbol $\#_i^d$;

(g) $$\delta_M(a, [q_i, A]) = (\#_0^0 \cdots \#_{j-1}^0 Q_j^0 \alpha, 1).$$

V. *Simulation of an accepting configuration*

(h) $$\delta_M(\#_k^d, B) = (\#_{k+1}^d \cdots \#_{n-1}^b, 1).$$

There are three basic types of moves that are made by machine M—(1) a move that is not an e-move, (2) an e-move in which M does not change states, and (3) an e-move that causes M to change state. The following claims verify that es machine E_M simulates each of these types of moves.

Claim 1. (not an e-move). If

$$(q_i, au, A\alpha) \vdash_{\overline{M}} (q_j, u, \beta\alpha),$$

then

$$(\#_i^d \cdots \#_{n-1}^b h_M(au), A\alpha B)$$

$$= (\#_i^d \cdots \#_{n-1}^b\, a\, \#_0^0 \cdots \#_{n-1}^b h_M(u), A\alpha B) \overset{*}{\underset{E_M}{\vdash}} (\#_j^0 \cdots \#_{n-1}^b h_M(u), \beta\alpha B).$$

Claim 1 follows directly from the definition of δ_M, parts (d), (e), (f), (g).

Claim 2. (e-move, no state change). If

$$(q_i, u, A\alpha) \underset{M}{\vdash} (q_i, u, \beta\alpha),$$

then

$$(\#_i^d \cdots \#_{n-1}^b h_M(u), A\alpha B) \underset{E_M}{\vdash} (\#_i^d \cdots \#_{n-1}^b h_M(u), \beta\alpha B).$$

Claim 2 follows from definition (b).

Claim 3. (e-move, changes state). If

$$(q_i, u, A\alpha) \underset{M}{\vdash} (q_j, u, \beta\alpha), \qquad q_i \neq q_j,$$

then for $d < b$

$$(\#_i^d \cdots \#_{n-1}^b h_M(u), A\alpha\beta) \overset{*}{\underset{E_M}{\vdash}} (\#_j^{d+1} \cdots \#_{n-1}^b h_M(u), \beta\alpha B).$$

Claim 3 follows from definitions (c), (d), (e).

We wish to show that $N(E_M) = w_M \cdot h_M[N(M)]$. Definition (a) places the bottom marker B on the store in order for E_M to know when M would have emptied its store. This is used in definition (k) for simulating an accepting configuration. Part (k) recognizes that M has empty store and merely reads any symbols remaining in the rightmost segment of w_M before popping the B from the store and accepting.

Claims 1 and 2 show that the simulation is well defined for e-moves that do not change state and for non-e-moves. The potential problem illustrated in Claim 3 is when we want to simulate an e-move that changes state and we are reading symbol $\#_i^b$.

This situation could only occur if we had already simulated b e-moves that caused a change of state. Since M changes state on consecutive e-moves ***at most b*** times, this situation cannot occur. Thus, the operation of simulating e-moves is also well defined. Therefore, we have verified that $N(E_M) = w_M \cdot h_M[N(m)]$.

Let us now return to our main problem of reducing the equivalence problem for languages in P to the equivalence problem for languages accepted by es machines. Let $M_i = (K_i, \Sigma, \Gamma_i, \delta_i, q_0, Z_0, F_i)$, $i = 1, 2$ be any two dpda's. Without loss of generality, we assume that $K_1 = K_2 = \{q_0, ..., q_{n-1}\}$ for some $n \geqslant 1$, and that there exists some $b \geqslant 0$ such that both M_1 and M_2 change state at most b times during consecutive e-moves. Then by the method outlined above, we can construct two es machines E_1 and E_2, such that for string $w = w_{M_1} = w_{M_2}$ and homomorphism $h = h_{M_1} = h_{M_2}$, we have

$$N(E_1) = w \cdot h[N(M_1)],$$
$$N(E_2) = w \cdot h[N(M_2)].$$

The homomorphism h is constructed so that for any two symbols $c_1, c_2 \in \Sigma$, $c_1 \neq c_2$,

(i) $h(c_1)$ is not a prefix of $h(c_2)$, and

(ii) $h(c_2)$ is not a prefix of $h(c_1)$.

Thus,

$$N(E_1) = N(E_2) \quad \text{iff} \quad N(M_1) = N(M_2).$$

Hence, we have proven the result. ▌

Corollary 3.3 and Theorem 3.4 show that the equivalence problem for languages in P is decidable iff it is decidable for extended simple languages. Recall the language

$$L = \{a^n b a^n b \mid n \geqslant 1\} \cup \{a^n c a^n c \mid n \geqslant 1\}.$$

Although L is not an es language, by inserting the proper number of symbols from $\{\#_i^d\}$, as indicated in Theorem 3.4 above, we obtain a language L' that is a homomorphic image of L, where L' is now accepted by an es machine. Thus, by giving our machines the ability of choosing whether or not to advance the input tape, we eliminate the need for having multiple states in a finite-state control. The insertion of markers between symbols in the original deterministic context-free language is not an unnatural request for actual computer languages—this is similar to the common addition of endmarkers at the right end of an input tape.

It is important to realize that the homorphism defined in the proof of Theorem 3.4 is of a very specialized nature: Not every homomorphic image of a deterministic context-free language is an es language. Indeed, it is well-known [2] that if L is a context-free language, then there exists a homomorphism h and a deterministic language L' such that $L = h(L')$. The characterization employed in the previous theorem uses a jdpda and a one-to-one homomorphism which is information lossless.

4. Monadic Recursion Schemes and Their Parallels with ES Languages

In this section we present definitions and terminology about monadic recursion schemes, and show how to translate between schemes and es machines. First, we provide a construction that when given a scheme S, produces an es machine that accepts the interpreted value language of S. Next, a construction is given such that for any es machine M, we are able to find a scheme with interpreted value language that "encodes" $N(M)$. Combining these translatability results with those of Section 3, we obtain the main result of this paper: The equivalence problem for deterministic languages is decidable if and only if the strong equivalence problem for monadic recursion schemes is decidable.

A *monadic recursion scheme* has a vocabulary consisting of a finite set of function symbols $\mathscr{F}$, a finite set of predicate symbols $\mathscr{P}$, and a finite set of function variable symbols $\mathscr{V}$ (with designated initial function variable V_0). A *term* is any string in $(\mathscr{F} \cup \mathscr{V})^*$. A *conditional term* is any expression of the form

$$\text{if } p \text{ then } \alpha \text{ else } \beta$$

where $p \in \mathscr{P}$, and α, β are terms or conditional terms. A monadic recursion scheme is a finite set of *function variable definitions* of the form

$$V \leftarrow \alpha,$$

α a term or conditional term, with exactly one for each function variable $V \in \mathscr{V}$.

The definition of how a scheme computes can be found in detail in [1, 3]. For this paper, we are only interested in computations relative to *free interpretations*, in which a function is interpreted as concatenation of its name. To define free interpretation I, we only need to specify a total function for each predicate symbol $p \in \mathscr{P}$

$$I(p) : \mathscr{F}^* \rightarrow \{1, 0\}.$$

The *value* of a scheme S under free interpretation I, denoted $\mathrm{val}_I(S)$, is the final value if the computation of S halts under interpretation I; it is undefined otherwise.

We use the relation $\vdash_{\overline{S,I}}$ as our notation corresponding to a single step in a computation of scheme S under interpretation I; $\overset{*}{\vdash}_{\overline{S,I}}$ is its transitive reflexive closure. Clearly

$$V_0 \overset{*}{\vdash}_{\overline{S,I}} \mathrm{val}_I(S).$$

Let S be a scheme with predicate symbols $\{p_1, ..., p_n\}$ and function symbols in set $\mathscr{F}$. We define the *interpreted value* of scheme S under free interpretation I, denoted $\mathrm{intval}_I(S)$, to be the string of $(n+1)$-tuples in $\{1, 0\}^n \times \mathscr{F}$ such that for new symbol $\$ \notin \mathscr{F}$

$$\mathrm{intval}_I(S) = [Q_1, f_1][Q_2, f_2] \cdots [Q_k, f_k][q_{k+1}, \$]$$

where

$$\mathrm{val}_I(S) = f_k \cdots f_1, \qquad f_1, ..., f_k \in \mathscr{F}$$

and for $1 \leqslant i \leqslant k+1$

$$Q_i = I(p_1)(f_{i-1} \cdots f_1), ..., I(p_n)(f_{i-1} \cdots f_1).$$

Q_i is an n-tuple representing the interpretation I for the n predicates on string $f_{i-1} \cdots f_1$ before concatenating function symbol f_i.

Define the *interpreted value language* for S to be

$$\mathrm{intval}(S) = \{\mathrm{intval}_I(S) \mid I \text{ is a free interpretation of } S \text{ and } \mathrm{intval}_I(S) \text{ is defined}\}.$$

Although it is not the usual one found in the literature [1, 3], the results of [3] allow us to make the following definition. Given two monadic recursion schemes S and T, S is said to be *strongly equivalent* to T, denoted $S \equiv T$, if and only if intval(S) $=$ intval(T). The question as to whether or not there is a decision procedure for determining if two schemes are strongly equivalent is known as the *strong equivalence problem.*

We now wish to establish precise relationships between monadic recursion schemes and es machines. We begin by showing that for any given monadic recursion scheme, we can construct an es machine that models its computational behavior. That is, the language accepted by the es machine encodes the scheme's computations on free interpretations.

4.1. THEOREM. *Given any monadic recursion scheme S, there is an effectively constructable* es *machine M such that $N(M) =$* intval(S).

Proof. Let S be a monadic recursion scheme with function symbols in set $\mathscr{F}$, function variable symbols in set $\mathscr{V}$, initial function variable symbol V_0. Without loss of generality, assume that the predicates in S are $\{p_1,..., p_n\}$, $n \geqslant 1$. Construct es machine $M = (\Sigma, \Gamma, \delta, Z_0)$ from scheme S, where $\Gamma = \mathscr{V} \cup \mathscr{F} \cup \{Z_0, B\}$, where B and Z_0 are two new symbols not in $\mathscr{V}$ or $\mathscr{F}$,

$$\Sigma = \{[d_1,..., d_n, f] \mid d_1,..., d_n \in \{1, 0\}, f \in \mathscr{F}\}$$
$$\cup \{[d_1,..., d_n, \$] \mid d_1,..., d_n \in \{1, 0\}\}.$$

For all $V \in \mathscr{V}, f \in \mathscr{F}$, $\alpha \in (\mathscr{V} \cup \mathscr{F})^*$, $d_1,..., d_n \in \{1, 0\}$, free interpretations I with $I(p_i)(e) = d_i$, $1 \leqslant i \leqslant n$, construct δ as follows: (α^R denotes the reversal of string α).

I. M simulates a step of computation of scheme S for some free interpretation I. (M advances its read head only if the term of S indicated by the interpretation has a function symbol as its rightmost symbol.)

(a) If $V \vdash_{\overline{S,I}} \alpha f$, then

$$\delta([d_1,..., d_n, f], V) = (\alpha^R, 1).$$

(b) If $V \vdash_{\overline{S,I}} \alpha$, $\alpha \notin (\mathscr{V} \cup \mathscr{F})^*\mathscr{F}$, then

$$\delta([d_1,..., d_n, f], V) = (\alpha^R, 0).$$

II. Initialize the es machine M. (Place bottom marker B on the store, and then simulate a step of a computation of scheme S as specified in I above.)

(c) If $V_0 \vdash_{\overline{S,I}} \alpha f$, then

$$\delta([d_1,..., d_n, f], Z_0) = (\alpha^R B, 1).$$

(d) If $V_0 \vdash_{\overline{S,I}} \alpha$, $\alpha \notin (\mathscr{V} \cup \mathscr{F})^*\mathscr{F}$, then

$$\delta([d_1,..., d_n, f], Z_0) = (\alpha^R B, 0).$$

III. Remove matching function symbols from the pushdown store;

(e) $$\delta([d_1, \ldots, d_n, f], f) = (e, 1).$$

IV. End Condition. The final symbol of an accepted string must be $[d_1, \ldots, d_n, \$]$, for some $d_1, \ldots, d_n \in \{1, 0\}$. (Recall that the rightmost symbol of each string in intval(S) is also of this form); pop the bottom marker B, if such a symbol is encountered when B is the only symbol in the store; if there is some pushdown symbol other than B on the top of the store when such an input symbol is read, then M can only operate with a non-advancing move; in addition, these nonadvancing moves can occur only when the next step of a computation of S would be an e-step.

(f) $$\delta([d_1, \ldots, d_n, \$], B) = (e, 1).$$

(g) $$\text{If } V \vdash_{S,I} \alpha,\ \alpha \notin (\mathscr{V} \cup \mathscr{F})^* \mathscr{F},$$
$$\text{then } \delta([d_1, \ldots, d_n, \$], V) = (\alpha^R, 0).$$

The construction above is fairly straightforward. Although the induction proof is omitted, we claim that for all $w \in \mathscr{F}^*$, free interpretations I,

$$V_0 \overset{*}{\vdash}_{S,I} w = \text{val}_I(S), \qquad \text{iff}$$

$$(\text{intval}_I(S), Z_0) \overset{*}{\vdash}_M (e, e), \qquad \text{iff}$$

$$\text{intval}_I(S) \in N(M).$$

Thus, $N(M) = \text{intval}(S)$. ▌

Corollary 4.2 below is a refinement of the result of Garland and Luckham [3] showing that the strong equivalence problem for monadic recursion schemes is reducible to the equivalence problem for deterministic languages.

4.2. Corollary. *The strong equivalence problem for monadic recursion schemes is reducible to the equivalence problem for* es *languages.*

Proof. Given any two monadic recursion schemes, S_1 and S_2 construct two es machines, M_1 and M_2, as outlined in the proof of Theorem 4.1 above so that

$$N(M_1) = \text{intval}(S_1) \qquad \text{and} \qquad N(M_2) = \text{intval}(S_2).$$

Hence,

$$S_1 \equiv S_2 \quad \text{iff} \quad \text{intval}(S_1) = \text{intval}(S_2) \quad \text{iff} \quad N(M_1) = N(M_2).$$

So far we have seen that the decidability of the equivalence problem for es languages implies the decidability of the strong equivalence problem for monadic recursion schemes (Corollary 4.2). Let us look at the converse to this result. The following theorem is a new result that provides the remaining key for establishing the relation between monadic recursion schemes and dpda's.

4.3. THEOREM. *The equivalence problem for es languages is reducible to the strong equivalence problem for monadic recursion schemes.*

Proof. Let $M = (\Sigma, \Gamma, \delta, Z_0)$ be an es machine. Wihout loss of generality, assume that $\Sigma = (a_1, ..., a_n)$, for some $n \geqslant 1$. (That is, impose some ordering on the elements in Σ.) We can also assume that if M ever empties the store, then the last move must have advanced the input (by merely marking the bottommost symbol).

Construct scheme S from M, with only one function symbol f, predicate symbols $\{p_1, ..., p_n\}$, function variable symbols $\Gamma \cup \{U\}$, $U \notin \Gamma$, initial variable symbol Z_0. The function variable definitions are constructed as follows.

(a) Loop: $U \to Uf$.

(b) For each $V \in \Gamma$, $\alpha_1, ..., \alpha_n \in \Gamma^*$,

$$V \leftarrow \text{if } p_1 \text{ then } \begin{Bmatrix} \alpha_1{}^R f, & \text{if } \delta(a_1, V) = (\alpha_1, 1), \\ \alpha_1{}^R, & \text{if } \delta(a_1, V) = (\alpha_1, 0), \\ Uf, & \text{if } \delta(a_1, V) \text{ is undefined,} \end{Bmatrix} \quad \text{else if}$$

$$p_2 \text{ then } \begin{Bmatrix} \alpha_2{}^R f, & \text{if } \delta(a_2, V) = (\alpha_2, 1), \\ \alpha_2{}^R, & \text{if } \delta(a_2, V) = (\alpha_2, 0) \\ UF, & \text{if } (a_1, V) \text{ is undefined,} \end{Bmatrix} \quad \text{else if}$$

$$\vdots$$

$$p_n \text{ then } \begin{Bmatrix} \alpha_n{}^R f, & \text{if } \delta(a_n, V) = (\alpha_n, 1), \\ \alpha_n, & \text{if } \delta(a_n, V) = (\alpha_n, 0), \\ Uf, & \text{if } \delta(a_n, V) \text{ is undefined,} \end{Bmatrix} \quad \text{else } Uf.$$

A computation of scheme S exactly parallels one of es machine M. It is not true, however, that $N(M) = \text{intval}(S)$, since $(N(M) \subseteq \Sigma^*) \neq (\text{intval}(S) \subseteq \{1, 0\}^n \times \{f, \$\})$ in general. But if we consider $a_i \in \Sigma$ as being "encoded" by symbols in $\{[d_1, ..., d_n, f] \mid d_1, ..., d_{i-1} = 0, d_i = 1, d_{i+1}, ..., d_n \in \{0, 1\}\}$, then intval$(S)$ precisely encodes $N(M)$. So for any string $x \in N(M)$, there is a free interpretation I such that $\text{intval}_I(S)$ encodes x. Consequently, for any two es machines M_1 and M_2, we can construct two schemes S_1 and S_2, from M_1 and M_2 respectively, as outlined above, where $N(M_1) = N(M_2)$ iff $S_1 \equiv S_2$.

We note that the assumption made in the first paragraph about how M empties its store is necessary for this construction. To illustrate this, consider the two machines M_1 and M_2 below, where M_1 empties the store on input tape a_2, without reading past this symbol.

$$M_1 : \delta(a_1, Z) = (e, 1), \qquad M_2 : \delta(a_1, Z) = (e, 1),$$
$$\delta(a_2, Z) = (e, 0), \qquad \delta(a_2, Z) = (Z, 0),$$
$$N(M_1) = N(M_2) = \{a_1\}.$$

The construction above gives us

$S_1 : Z_0 \leftarrow$ if p_1 then f else if p_2 then e else Uf,

$U \leftarrow Uf$,

$S_2 : Z_0 \leftarrow$ if p_1 then f else if p_2 then Z else Uf,

$U \leftarrow Uf$,

$S_1 \not\equiv S_2$ since $[0, 1, \$] \in \mathrm{intval}(S_1)$, but $(0, 1, \$] \notin \mathrm{intval}(S_2)$.

Notice that the construction of the scheme S in the proof of Theorem 4.3 is such that it uses only one function symbol, f. The following result is an immediate consequence.

4.4. COROLLARY. *The equivalence problem for* es *languages is reducible to the strong problem for monadic recursion schemes that use only one function symbol.*

Another important corollary follows from the proof of Theorem 4.3. In fact, Corollary 4.5 below is the main result in this paper. As stated earlier, our aim in this paper was to show that given any two dpda's, M_1 and M_2, one can effectively construct two monadic recursion schemes, S_1 and S_2, such that $T(M_1) = T(M_2)$ iff $S_1 \equiv S_2$. Now, we prove that this is indeed the case.

4.5. COROLLARY. *The equivalence problem for deterministic context-free languages is decidable if and only if the strong equivalence problem for monadic recursion schemes is decidable.*

Proof. By using endmarkers, we need only consider the equivalence problem for prefix-free deterministic context-free languages, P, instead of the larger family D. The rest of the proof follows from Corollary 3.3, Theorem 3.4, Corollary 4.2, and Theorem 4.3. ∎

5. CONCLUSIONS

We have extended the result of Garland and Luckham [3] to show that the strong equivalence problem for monadic recursion schemes is decidable if and only if the equivalence problem for deterministic context-free languages is decidable. This result was accomplished by encoding each deterministic context-free languages into a "simpler" language referred to as an extended simple language. We have shown that although the es languages are properly contained in the deterministic context-free languages, their equivalence problems are equivalent. We hope that the single-state feature of the es languages will provide some insight into proving the decidability of the equivalence problem for the deterministic context-free languages.

Acknowledgment

I would like to thank Bruno Courcelle for his helpful comments on an earlier version of Theorem 3.4.

References

1. E. Ashcroft, A. Manna, and A. Pnueli, Decidable properties of monadic functional schemes, *J. Assoc. Comput. Mach.* **20** (1973), 489–499.
2. N. Chomsky and M. Schutzenberger, The algebraic theory of context-free languages, *in* "Computer Programming and Formal Systems" (P. Braffort and D. Hirschberg, Eds.), North-Holland, Amsterdam, 1963.
3. S. J. Garland and D. C. Luckham, Program schemes, recursion schemes, and formal languages, *J. Comput. System Sci.* **7** (1973), 119–160.
4. S. Ginsburg and S. Greibach, Deterministic context-free languages, *Inform. Contr.* **9** (1966), 620–648.
5. S. A. Greibach, Jump pda's and hierarchies of deterministic context-free languages, *SIAM J. Comput.* **3** (1974), 11–127.
6. M. A. Harrison and I. M. Havel, Strict deterministic context-free languages, *J. Comput. System Sci.* **7** (1973), 237–277.
7. A. J. Korenjak and J. E. Hopcroft, Simple deterministic languages, *in* "Proceedings of the Seventh Annual IEEE Symposium on Switching and Automata Theory," pp. 36–46, Berkeley, Calif., 1966.
8. A. L. Rosenberg, Real-time definable languages, *J. Assoc. Comput. Mach.* **14** (1967), 645–662.