INFORMATION AND COMPUTATION 95, 21–75 (1991)

Iterated Stack Automata and Complexity Classes

JOOST ENGELFRIET

Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands

An iterated pushdown is a pushdown of pushdowns of ... of pushdowns. An iterated exponential function is 2 to the 2 to the ... to the 2 to some polynomial. The main result presented here is that the nondeterministic 2-way and multi-head iterated pushdown automata characterize the deterministic iterated exponential time complexity classes. This is proved by investigating both nondeterministic and alternating auxiliary iterated pushdown automata, for which similar characterization results are given. In particular it is shown that alternation corresponds to one more iteration of pushdowns. These results are applied to the 1-way iterated pushdown automata: (1) they form a proper hierarchy with respect to the number of iterations, and (2) their emptiness problem is complete in deterministic iterated exponential time. Similar results are given for iterated stack (checking stack, nonerasing stack, nested stack, checking stack-pushdown) automata.

INTRODUCTION

It is well known that several types of 2-way and multi-head pushdown automata and stack automata have the same power as certain time or space bounded Turing machines; see, e.g., Chapter 14 of (Hopcroft and Ullman, 1979), or Sections 13 and 20.2 of (Wagner and Wechsung, 1986). For the deterministic and nondeterministic case such characterizations were given by Fischer (1969) for checking stack automata, by Hopcroft and Ullman (1967b) for nonerasing stack automata, by Cook (1971) for auxiliary pushdown and 2-way stack automata, by Ibarra (1971) for auxiliary (nonerasing and erasing) stack automata, by Beeri (1975) for 2-way and auxiliary nested stack automata, and by van Leeuwen (1976) for auxiliary checking stack-pushdown automata. The alternating case was considered by Chandra, Kozen, and Stockmeyer (1981), and was extensively studied by Ladner, Lipton, and Stockmeyer (1984). The highest complexity class reached by the 2-way or multi-head versions of these automata is double exponential time: the class of languages recognized by alternating multi-head stack automata (Ladner, Lipton, and Stockmeyer, 1984). Higher complexity classes were characterized by Vogel and Wagner (1985). Let $\exp_k(n)$ be the k-iterated exponential function, where k is the number

0890-5401/91 $3.00

of 2's: $\exp_0(n) = n$, and $\exp_k(n+1) = 2^{\exp_k(n)}$. It is shown in (Vogel and Wagner, 1985) that DTIME($\exp_k$(poly)) is the class of languages accepted by deterministic multi-head automata with one pushdown and k (independent) checking stacks. Similarly, for a nonerasing stack instead of a pushdown, the class DSPACE($\exp_k$(poly)) is obtained. In both cases the corresponding nondeterministic automata of course accept all recursively enumerable languages.

In this paper we present an alternative automaton-theoretic characterization of DTIME($\exp_k$(poly)), in terms of the nondeterministic multi-head $(k+1)$-iterated pushdown automata (where a 1-iterated pushdown is an ordinary pushdown, a 2-iterated pushdown is a pushdown of pushdowns, etc.). According to Greibach (1970), iterated pushdown automata were first considered by Aho and Ullman: they showed that the nondeterministic 1-way 2-iterated pushdown automata recognize the indexed languages of (Aho, 1968); see (Parchmann, Duske, and Specht, 1980) for essentially the same result. Greibach (1970) shows how pushdowns can be iterated by the use of "nested AFA," but only the special case of "well-nested AFA" is studied there. Maslov (1974, 1976) defines the nondeterministic 1-way k-iterated pushdown automata, and shows that they correspond to the k-level indexed grammars. Damm and Goerdt (1986) prove that they correspond to the k-level OI macro grammars. The classes of k-level OI languages (and hence the classes of nondeterministic 1-way k-iterated pushdown languages) are often viewed as an alternative, more natural, infinite, Chomsky hierarchy, called the OI-hierarchy; see, e.g., (Damm, 1982). Taking a 0-iterated pushdown automaton to mean a finite automaton, the first three classes in the OI-hierarchy consist of the regular, the context-free, and the indexed languages.

Our main results on k-iterated pushdown automata (P^k automata) are given in the table of Fig. 1, where we consider nondeterministic or alternating P^k automata which may be 1-way, (2-way) r-head with $r \geqslant 1$, (2-way) multi-head, or SPACE($s(n)$) auxiliary P^k automata with $s(n) \geqslant \log n$. Note that the multi-head case follows from both the r-head and the SPACE($\log n$) case; it is added for clearness sake. For $k = 1$ ($k = 0$), the results are of course the known ones for pushdown automata (finite automata, respectively). For $k = 2$, the results are those for stack automata: we show that the mentioned types of 2-iterated pushdown automata are equivalent to the corresponding stack automata. Since, moreover, almost all reasonable types of 2-iterated pushdown automata are equivalent to the corresponding nested stack automata (e.g., the 1-way types both recognize the indexed languages), this also fits with the results of (Beeri, 1975). In fact, for iterated stack automata (SA^k) and iterated nested stack automata (NSA^k) we show that the table of Fig. 1 holds with k replaced by $2k$ everywhere. These complexity characterizations are applied to the 1-way P^k

P^k	nondeterministic	alternating
1-way	-----	$\cup$DTIME($\exp_k(dn)$) $k \geq 1$
r-head	$\cup$DTIME($\exp_{k-1}(dn^{2r})$) $k \geq 2$	$\cup$DTIME($\exp_k(dn^r)$) $k \geq 1$
multi-head	DTIME($\exp_{k-1}$(poly)) $k \geq 1$	DTIME($\exp_k$(poly)) $k \geq 0$
SPACE(s(n))	$\cup$DTIME($\exp_k(ds(n))$) $k \geq 1$	$\cup$DTIME($\exp_{k+1}(ds(n))$) $k \geq 0$

FIG. 1. Characterization of k-iterated pushdown automata by time-bounded Turing machines. In this table, $\cup \ldots d \ldots$ abbreviates $\cup_{d>0} \ldots d \ldots$, and $\ldots$ poly $\ldots$ abbreviates $\cup_{d>0} \ldots n^d \ldots$.

automata: (1) they form a proper hierarchy, i.e., the $(k+1)$-iterated pushdown automata are more powerful than the k-iterated pushdown automata, and (2) their emptiness problem is complete in DTIME($\exp_{k-1}$(poly)). Similar results hold for SA^k and NSA^k automata.

Additionally we investigate (iterated) checking stack and nonerasing stack automata, and stack-pushdown automata; for the latter, see (van Leeuwen, 1976, Engelfriet, Schmidt, and van Leeuwen, 1980). In particular, iterated checking stack automata characterize iterated exponential space (rather than time) complexity classes. Since in the 1-way case these automata correspond to the 2GSM hierarchy (see Greibach, 1978c, Engelfriet, 1982), this gives an alternative proof of the properness of that hierarchy. It also implies that the emptiness problem for arbitrary compositions of 2-way gsm's is nonelementary (i.e., is not in $\bigcup_k$ DTIME($\exp_k$(poly)). A similar relationship holds between the iterated checking stack-pushdown automata and the ETOL hierarchy; see (Asveld and van Leeuwen, 1975, Engelfriet, 1982).

A preliminary version of this paper was presented in (Engelfriet, 1983). It was recently shown in (Kowalczyk, Niwinski, and Tiuryn, 1989) that deterministic SPACE($s(n)$) auxiliary P^k automata have the same power as nondeterministic ones (for space constructable $s(n)$), generalizing the result of (Cook, 1971) for $k = 1$; this problem was left open in (Engelfriet, 1983).

NOTATION

For a set A, $P_{\text{fin}}(A)$ denotes the set of all finite subsets of A. For a relation R, R^{-1} is its inverse and $R(A)$ is the image of A under R. For a class S of relations and a class K of sets, $S^{-1} = \{R^{-1} \mid R \in S\}$, and $S(K) = \{R(A) \mid R \in S, A \in K\}$. $S^0(K) = K$, and for $k \geqslant 0$, $S^{k+1}(K) = S(S^k(K))$.

For a set A, A^* is the set of finite sequences of elements of A, and $A^+ = A^* - \{\lambda\}$, where λ is the empty sequence. In case A is an alphabet, A^* is the set of strings over A, and λ is the empty string. Let $w = a_1 a_2 \cdots a_n$ be a string with $a_i \in A$. Then $|w|$ is its length n. The reverse of w is the string $w^R = a_n \cdots a_2 a_1$. For a language L, $L^R = \{w^R \mid w \in L\}$, and for a family K of languages $K^R = \{L^R \mid L \in K\}$. REG denotes the family of regular languages.

We use $\bigcup \ldots d \ldots$ for $\bigcup_{d>0} \ldots d \ldots$, and $\ldots \text{poly} \ldots$ for $\bigcup_{d>0} \ldots n^d \ldots$ (as in Fig. 1).

For the notation of time and space complexity classes, see Chapter 12 of (Hopcroft and Ullman, 1979).

1. BASIC DEFINITIONS AND FACTS

In this section we describe the iterated pushdown automata which are the main subject of this paper. Since we will consider many variations of iterated pushdown automata, and also of other automata, we need some general terminology on automata with an arbitrary storage type, explained in Section 1.1. In order to cope with essentially equivalent ways of describing the same kind of automaton, we also discuss the notion of equivalent storage types, in Section 1.2. In Section 1.3, finally, the iterated pushdown automata are defined. In fact, for any storage type X (such as pushdown, stack, etc.), we consider pushdown-of-X automata, i.e., automata of which the storage consists of a pushdown of storage configurations of an X automaton. Our main technique in dealing with the rather complex iterated pushdown automata is to generalize known results for pushdown automata to pushdown-of-X automata, for arbitrary X, and then to iterate the "pushdown-of" operation, in order to obtain similar results for iterated pushdown automata. The same approach was used in (Engelfriet and Vogler, 1987) for deterministic 1-way P^k automata, in (Engelfriet and Vogler, 1988) for iterated pushdown tree transducers, and in (Vogler, 1986) for iterated one-turn pushdown automata.

1.1. *Automata and Storage Types*

Let X be a storage type of an automaton (e.g., $X =$ pushdown). We assume the reader to be familiar with

1-way X automata,

2-way multi-head X automata, and

auxiliary SPACE($s(n)$) X automata (for s: $\mathbb{N} \to \mathbb{N}$),

and with the corresponding automata with output, i.e., transducers. Recall that an auxiliary SPACE($s(n)$) X automaton has, in addition to its X-storage, a 2-way read-only input tape and a Turing machine worktape with space restricted to $s(n)$, where n is the length of the input. The above-mentioned automata may be deterministic, nondeterministic, or alternating (except that transducers cannot be alternating). All storage types X investigated in this paper (such as X = pushdown, k-iterated pushdown, stack, checking stack, etc.) will be explained in due course; for more details on the corresponding X automata see, e.g., Hopcroft and Ullman (1979), Cook (1971), Ladner, Lipton, and Stockmeyer (1984), and Ibarra (1971). In the special case that there is no internal storage X, we put "finite" instead of X, i.e., we have the usual 1-way finite automata and 2-way multi-head finite automata. The auxiliary SPACE($s(n)$) finite automata are of course the usual $s(n)$-space bounded Turing machines.

Notation. The class of languages accepted by alternating 1-way (2-way r-head, 2-way multi-head, auxiliary SPACE($s(n)$)) X automata is denoted $1A-X$ ($2A(r)-X$, $2A(\text{multi})-X$, $ASPACE(s(n))-X$, respectively), where $r \geqslant 1$ and s: $\mathbb{N} \to \mathbb{N}$. For the deterministic and nondeterministic automata, the leading A is replaced by D and N, respectively. The class of transductions realized by nondeterministic (deterministic) 1-way X transducers is denoted $1NT-X$ ($1DT-X$, respectively), and similarly for the other variations (adding T before "$-X$"). In case X = "finite", the suffix "$-X$" is dropped; thus $2D(r)$ denotes the class of languages accepted by 2-way deterministic r-head finite automata, and NSPACE($s(n)$), as usual, the class of languages accepted by nondeterministic SPACE($s(n)$) Turing machines. We finally note that the heads of a 2-way r-head automaton may be sensing or non-sensing: our results do not depend on this distinction.

We will use Y as a metavariable ranging over the set {1A, 2A(r), 2A(multi), ASPACE($s(n)$), 1N, ..., 1NT, ..., 1D, ..., 1DT, ...} of all variations of X automata mentioned above. An element of this set will be called an *automaton type*. Thus, for an automaton type Y and a storage type X, we may consider the class of $Y-X$ automata, and the corresponding class $Y-X$ of languages (or transductions).

For the investigation of X automata (or, rather, pushdown-of-X automata) for arbitrary storage type X, we need some precise terminology on storage types. Our definition of storage type (as in Engelfriet and Vogler (1987); see also Engelfriet (1986), Engelfriet and Vogler (1986, 1988),

Engelfriet and Hoogeboom (1989)) is based on those of Ginsburg (1975), Scott (1967), Hopcroft and Ullman (1967a), and Goldstine (1977).

DEFINITION. A *storage type* is a tuple $X=(C, T, F, m, C_0, \mathrm{id})$, where C is the set of (X-) configurations, $C_0 \subseteq C$ is the set of initial configurations, T is the set of tests, F is the set of instructions, $\mathrm{id} \in F$ is the identity instruction, and m is the meaning function that associates with every $t \in T$ a mapping $m(t)$: $C \to \{\text{true}, \text{false}\}$, and with every $f \in F$ a partial function $m(f)$: $C \to C$, such that $m(\mathrm{id})$ is the identity on C.

The sets C, T, F, and C_0 may all be infinite.

A test t or instruction f is "executed" by an X automaton by applying the function $m(t)$ or $m(f)$, respectively, to its X-configuration. Note that the execution of id has no effect on storage (it may be pronounced as "idle"). To model the execution of several instructions $f_1, \ldots, f_n \in F$, in that order, we extend m to F^+, by defining $m(f_1 f_2 \cdots f_n)(c) = m(f_n)(\cdots m(f_2)(m(f_1)(c)) \cdots)$ for every $c \in C$.

As an example, the usual *pusdown* storage type P is defined by taking $C = \Gamma^+$ for some fixed, infinite, set of pushdown symbols, $C_0 = \Gamma$, $T = \{\mathrm{top}=\gamma \mid \gamma \in \Gamma\}$ such that $m(\mathrm{top}=\gamma)(w) = \text{true}$ iff the right-most symbol of $w \in \Gamma^+$ is γ, and $F = \{\mathrm{push}(\gamma) \mid \gamma \in \Gamma\} \cup \{\mathrm{pop}, \mathrm{id}\}$ such that $m(\mathrm{push}(\gamma)) = \{(w, w\gamma) \mid w \in \Gamma^+\}$, $m(\mathrm{pop}) = \{(w\gamma, w) \mid w \in \Gamma^+, \gamma \in \Gamma\}$, and $m(\mathrm{id}) = \{(w, w) \mid w \in \Gamma^+\}$, Note that there is no empty pushdown; this simplifies things when generalizing to pushdown-of-X.

To be able to treat automata without internal storage as a special case of X automata (i.e., $X =$ "finite"), we use the *trivial storage type*, denoted X_0 and defined by $X_0 = (\{c_0\}, \varnothing, \{\mathrm{id}\}, m, \{c_0\}, \mathrm{id})$, where c_0 is an arbitrary object and $m(\mathrm{id})$ is the identity on $\{c_0\}$. Thus, X_0 automata are finite automata (and the suffix "$-X_0$" can be dropped, see Notation).

In the rest of this subsection we want to fix some more terminology on the way $Y-X$ automata work, and in particular the way they manipulate storage, so that the reader can check that everything is as usual. Let $X=(C, T, F, m, C_0, \mathrm{id})$ be a storage type.

We first need a notion formalizing the test information that an automaton can obtain from its storage. For a finite subset T_1 of T, the set of *test results* for T_1, denoted $R(T_1)$, is the set $R(T_1) = \{\rho \mid \rho \text{ is a mapping } T_1 \to \{\text{true}, \text{false}\}\}$. For $c \in C$, the *test result of* c for T_1, denoted by $\rho(T_1, c)$, is the element of $R(T_1)$ that satisfies $\rho(T_1, c)(t) = m(t)(c)$ for every $t \in T_1$. Note that $R(\varnothing)$ is a singleton.

Next we discuss the specification of X automata. Let Y be an automaton type. A $Y-X$ automaton M is specified by a set of states Q (divided into existential and universal states in case M is alternating), an initial state $q_0 \in Q$ and a set of final states $Q_H \subseteq Q$, an input alphabet Σ, an output

alphabet Ω (in case M is a transducer), a worktape alphabet Φ (in case M is an auxiliary automaton), a transition function δ (the most important part), an initial X-configuration $c_0 \in C_0$, a finite subset T_M of T, and a finite subset F_M of F (the tests and instructions used by M, respectively). The transition function δ is a mapping of the form

$$\delta: A \times R(T_M) \to P_{\text{fin}}(B \times F_M)$$

where $A = Q \times A'$ and $B = Q \times B'$, and the precise form of the sets A' and B' depends on the precise type Y of the automaton M. If M is a 1-way X automaton, then $A' = \Sigma \cup \{\lambda\}$ and B' is trivial, i.e., δ: $Q \times (\Sigma \cup \{\lambda\}) \times R(T_M) \to P_{\text{fin}}(Q \times F_M)$. In case it is a transducer, $B' = \Omega^*$, i.e., δ maps into $P_{\text{fin}}(Q \times \Omega^* \times F_M)$. If M is a 2-way r-head automaton, then $A' = (\Sigma \cup \{¢, \$\})^r$ and $B' = (-1, 0, +1\}^r$, where ¢ and \$ are the left and right endmarkers, respectively, and -1, 0, $+1$ indicate the motions of the heads, as usual (note that here we assume for convenience that the heads are non-sensing; this is, however, not essential). If M is an auxiliary SPACE($s(n)$) X automaton, then $A' = (\Sigma \cup \{¢, \$\}) \times \Phi$ and $B' = \{-1, 0, +1\} \times \{-1, 0, +1\} \times \Phi$; note that M has one input head and one worktape head. For the corresponding transducers, Ω^* is added to B'.

In the 2-way multi-head case and the auxiliary SPACE($s(n)$) case, M is *deterministic* if δ is a partial function, i.e., $\delta(x)$ is a singleton or empty for every x in its domain. In the 1-way case it is additionally required that if $\delta(q, \lambda, \rho) \neq \varnothing$ for some $q \in Q$ and $\rho \in R(T_M)$, then $\delta(q, \sigma, \rho) = \varnothing$ for all $\sigma \in \Sigma$ (in words, if M can do a *λ-move*, then it cannot do a reading move).

Finally we discuss the computations of our automaton M. An instantaneous description (ID) of M is a pair (α, c), where $c \in C$ is an X-configuration, and α consists of the usual things: a state $q \in Q$, an input string $w \in \Sigma^*$, the position of the input head(s) on w (or on ¢w\$), possibly an output string in Ω^*, and possibly a worktape string in Φ^* together with the position of the worktape head. If M is alternating, then (α, c) is a universal (existential) ID if q is a universal (existential, respectively) state. Let (α, c) and (β, d) be two ID's of M where $c, d \in C$, and α, β represent the rest of the ID's. Suppose that $\delta(a, \rho)$ contains (b, f) with $a \in A$, $\rho \in R(T_M)$, $b \in B$, and $f \in F_M$. Then M can do the move $(\alpha, c) \vdash (\beta, d)$ if α satisfies a (in the usual way), $\rho = \rho(T_M, c)$, the test result of c for T_M, β is obtained from α by b (in the usual way), $m(f)(c)$ is defined, and $m(f)(c) = d$. (β, d) is called a *successor* of (α, c). As usual, $\vdash^*$ denotes the transitive, reflexive closure of $\vdash$.

The initial ID of M for an input string $w \in \Sigma^*$ is (α_0, c_0) where c_0 is the initial X-configuration of M, and α_0 consists of the initial state q_0, the input string w, all input heads positioned to the left of w, an empty output string,

and a blank worktape. Acceptance is by final state, i.e., an accepting ID of M is an ID of which the state belongs to Q_H (thus there are no requirements on the X-configuration of the ID).

For a nondeterministic (or deterministic) automaton M, a string $w \in \Sigma^*$ is in the *language* $L(M)$ *accepted* by M if there is a *computation*, i.e., a finite sequence of consecutive moves of M, that starts with the initial ID for w and ends with an accepting ID. In other words, $w \in L(M)$ iff $(\alpha_0, c_0) \vdash^* (\alpha, c)$, where (α_0, c_0) is the initial ID for w and (α, c) is an accepting ID. Moreover, for an auxiliary SPACE($s(n)$) automaton all worktape strings of all ID's in the computation should be of length at most $s(n)$, where $n = |w|$. Similarly, for a transducer, $(w, v) \in \Sigma^* \times \Omega^*$ is in the *transduction* $\tau(M)$ *realized* by M if moreover the accepting ID contains v as output string. Note that, for a deterministic transducer M, $\tau(M)$ is not necessarily a partial function. For an alternating automaton M, a *computation tree* is a finite tree of which the nodes are labeled by ID's of M, such that the children of a nonleaf labeled by a universal (existential) ID are labeled by all successors (one successor, respectively) of that ID (and, as above, in the auxiliary case all ID's in the tree should satisfy the corresponding length restriction). A string $w \in \Sigma^*$ is in the language $L(M)$ accepted by M if there is a computation tree of which the root is labeled by the initial ID for w, and all leaves are labeled by accepting ID's.

The class of all languages accepted (or translations realized) by $Y-X$ automata is denoted $Y-X$.

Two automata M and M' are equivalent if $L(M) = L(M')$, or $\tau(M) = \tau(M')$ if they are transducers.

1.2. *Simulation and Equivalence of Storage Types*

This subsection can be glanced at on first reading; it is only necessary to read Definition 1.2.1 (of $\leqslant$ and $\equiv$) and to believe Theorem 1.2.4.

One of the basic notions related to data types is that of one data type simulating another: e.g., a pushdown of integers can be simulated by an array of integers, in the usual way, and a similar statement of course holds for a pushdown of booleans. Since a storage type of an automaton is a data type, the same notion of simulation applies to storage types: e.g., a Turing machine tape can be simulated by two pushdowns (and vice versa).

We will need a formal definition of this concept of simulation. In Engelfriet and Vogler (1986) such a definition was given, based on the idea of stepwise simulation of Hoare (1972): roughly, each instruction or test of the first storage type is replaced by a procedure that simulates it in the second storage type. Although this definition is intuitively clear, it is quite long, and difficult to handle. Therefore, in this paper (and in Engelfriet and Hoogeboom, 1989), we propose a more manageable definition. It is weaker

than the one in (Engelfriet and Vogler, 1986), but can serve the same purposes, for the restricted kind of storage type we have defined here.

DEFINITION 1.2.1. Let X_1 and X_2 be two storage types. X_2 *simulates* X_1, denoted $X_1 \leqslant X_2$, if $1\mathrm{DT}-X_1 \subseteq 1\mathrm{DT}-X_2$. X_1 and X_2 are *equivalent*, denoted $X_1 \equiv X_2$, if $1\mathrm{DT}-X_1 = 1\mathrm{DT}-X_2$.

Obviously, the simulation relation $\leqslant$ is reflexive and transitive; moreover, $X_1 \equiv X_2$ if and only if $X_1 \leqslant X_2$ and $X_2 \leqslant X_1$.

In words, two storage types are equivalent if they define the same class of 1-way deterministic transductions. Now this is certainly something one would require of (intuitively) equivalent storage types, but we claim that it also suffices. One way to convince the reader of this unprovable claim is to prove that if $X_1 \equiv X_2$ then $Y-X_1 = Y-X_2$ for every automaton type Y (which one would also expect of intuitively equivalent storage types). The proof of this will be based on another, intuitive, argument that the above definition captures the intuitive notion of equivalence: intuitively a storage type may be viewed as a transduction (for a given fixed initial configuration), as follows. Suppose M is an X automaton with its storage in a certain configuration, reached from the initial configuration. What can M do with its storage? At one of its moves, M feeds an instruction into storage, as a result of which storage changes configuration. Then M obtains from storage the test result of the new configuration, upon which the choice of its next move will be based. Thus, storage receives information, changes configuration, and returns information. In this way storage acts as a transducer. We now formalize the corresponding transductions. They are analogous to the AFL generators obtained from an AFA representation of an AFL (see Section 5.2 of Ginsburg, 1975).

DEFINITION 1.2.2. Let $X=(C, T, F, m, C_0, \mathrm{id})$ be a storage type. Let T_1 and F_1 be finite subsets of T and F, respectively, and let $c_0 \in C_0$ be an initial X-configuration. The *X-transduction* corresponding to T_1, F_1, and c_0, denoted $\tau(T_1, F_1, c_0)$, is the relation in $F_1^* \times R(T_1)^*$ defined by $\tau(T_1, F_1, c_0) = \{(f_1 \cdots f_k, \rho_1 \cdots \rho_k) \mid k \geqslant 0,\ f_i \in F_1,\ \rho_i \in R(T_1),\ m(f_1 \cdots f_k)(c_0)$ is defined, and $\rho_i = \rho(T_1, m(f_1 \cdots f_i)(c_0))$ for every i, $1 \leqslant i \leqslant k\}$.

In fact $\tau(T_1, F_1, c_0)$ is a partial function $F_1^* \to R(T_1)^*$. Its domain is $\{f_1 \cdots f_k \mid m(f_1 \cdots f_k)(c_0)$ is defined$\}$. Note that if $m(f_1 \cdots f_k)(c_0)$ is defined, then so is $m(f_1 \cdots f_i)(c_0)$ for $1 \leqslant i \leqslant k$. When $f_1, f_2, \ldots, f_k$ are "fed," one by one, "into" $\tau(T_1, F_1, c_0)$, it "produces," one by one, the test results of $m(f_1)(c_0)$, $m(f_1 f_2)(c_0), \ldots, m(f_1 f_2 \cdots f_k)(c_0)$ for T_1. We now show that these X-transductions are in $1\mathrm{DT}-X$.

LEMMA 1.2.3. *Let* $X=(C, T, F, m, C_0, \mathrm{id})$ *be a storage type. For every* $T_1 \subseteq T$, $F_1 \subseteq F$ *(both finite), and* $c_0 \in C_0$, $\tau(T_1, F_1, c_0) \in \mathrm{1DT}-X$.

Proof. The 1-way deterministic X transducer M that realizes $\tau(T_1, F_1, c_0)$ has initial configuration c_0, and uses $T_M = T_1$ and $F_M = F_1$. M, when reading $f \in F_1$, executes f and then outputs the test result of the resulting X-configuration. Formally, M has transition function δ: $Q \times (\Sigma \cup \{\lambda\}) \times R(T_1) \to Q \times \Omega^* \times F_1$, with $Q=\{q_1, q_2\}$, and $\delta(q_1, f, \rho) = (q_2, \lambda, f)$, $\delta(q_2, \lambda, \rho) = (q_1, \rho, \mathrm{id})$ for all $\rho \in R(T_1)$ and $f \in F_1$, where q_1 is the initial state and the only final state. Note that $\Sigma = F_1$ and $\Omega = R(T_1)$. ∎

We use the X-transductions in the proof of the following "Justification Theorem" (it justifies our definition of equivalent storage types; see also Corollary 3.9 of (Engelfriet and Hoogeboom, 1989) and Theorem 4.18 of (Engelfriet and Vogler, 1986)).

THEOREM 1.2.4. *Let* X_1 *and* X_2 *be storage types, and let* Y *be an automaton type.*

If $X_1 \leqslant X_2$, *then* $Y-X_1 \subseteq Y-X_2$.

If $X_1 \equiv X_2$, *then* $Y-X_1 = Y-X_2$.

Proof. Obviously the second statement follows from the first. Let $X_i = (C_i, T_i, F_i, m_i, C_{0i}, \mathrm{id}_i)$ for $i=1, 2$, and assume that $X_1 \leqslant X_2$. We have to show that $Y-X_1 \subseteq Y-X_2$. For $Y=\mathrm{1DT}$ this holds by definition of $\leqslant$. We now give the proof simultaneously for all other Y. Let M be a $Y-X_1$ automaton with initial configuration $c_M \in C_{01}$, set of tests $T_M \subseteq T_1$, and set of instructions $F_M \subseteq F_1$. The use that M makes of its storage is fully determined by the X_1-transduction $\tau = \tau(T_M, F_M, c_M)$. Since $\tau \in \mathrm{1DT}-X_1$ (by Lemma 1.2.3) and $X_1 \leqslant X_2$, also $\tau \in \mathrm{1DT}-X_2$. Let N be a 1-way deterministic X_2 transducer realizing τ. We have to show the existence of a $Y-X_2$ automaton M' equivalent to M. Intuitively, M' is obtained from M and N by a variation of the usual product construction. In fact M' imitates the behaviour of M, using N instead of X_1-storage. Whenever M executes an instruction f of F_1, M' instead feeds f into N, simulates N until it produces a test result ρ, and stores ρ in its finite control, using it to simulate the next move of M.

Formally, let δ_M: $Q_M \times A' \times R(T_M) \to P_{\mathrm{fin}}(Q_M \times B' \times F_M)$ and δ_N: $Q_N \times (F_M \cup \{\lambda\}) \times R(T_N) \to Q_N \times R(T_M)^* \times F_N$ be the transition functions of M and N, respectively, where Q_M and Q_N are the set of states of M and N, respectively, and T_N and F_N are the finite subsets of T_2 and F_2 used by N. The precise form of A' and B' depends on Y.

Clearly we may assume that N produces at most one symbol at a time. Moreover, it is not difficult to argue that we may also assume that in any

successful computation of N, translating $f_1 \cdots f_k$ into $\rho_1 \cdots \rho_k$, ρ_i is not produced before f_i is read.

We now describe M'. It has the same initial configuration as N, and it also uses T_N and F_N. The set of states of M' is $Q = Q_M \times Q_N \times R(T_M)$, and the initial state is $\langle q_0, p_0, \rho_0 \rangle$ where q_0 and p_0 are the initial states of M and N, respectively, and $\rho_0 = \rho(T_M, c_M)$. A state $\langle q, p, \rho \rangle$ of M' is final whenever q and p are. In case M is alternating, a state $\langle q, p, \rho \rangle$ is existential or universal whenever q is. M' has the same input (worktape, output) alphabet as M. It remains to specify the transition function δ: $Q \times A' \times R(T_N) \to P_{\text{fin}}(Q \times B' \times F_N)$ of M'. Let $b'_0 \in B'$ be such that it leaves the involved devices invariant (i.e., no output, no heads moving, etc.). Moreover, in what follows, $q, q' \in Q_M$, $p, p' \in Q_N$, $\rho, \rho' \in R(T_M)$, $f \in F_M$, $\beta \in R(T_N)$, $g \in F_N$, $a' \in A'$, and $b' \in B'$. First we treat the λ-moves of N.

— If $\delta_N(p, \lambda, \beta) = (p', \lambda, g)$, then $\delta(\langle q, p, \rho \rangle, a', \beta) = \{(\langle q, p', \rho \rangle, b'_0, g)\}$.

— Similarly, if $\delta_N(p, \lambda, \beta) = (p', \rho', g)$, then the new state of M' is $\langle q, p', \rho' \rangle$.

Second we treat the other moves of N, together with the moves of M.

— If $\delta_N(p, f, \beta) = (p', \lambda, g)$ and $\delta_M(q, a', \rho)$ contains (q', b', f), then $\delta(\langle q, p, \rho \rangle, a', \beta)$ contains $(\langle q', p', \rho \rangle, b', g)$.

— Similarly, again, if $\delta_N(p, f, \beta) = (p', \rho', g)$, then the new state of M' is $\langle q', p', \rho' \rangle$.

This ends the description of M'. It easily follows from this description that M' is deterministic if M is.

In a state $\langle q, p, \rho \rangle$, ρ represents the test result of the configuration of M. After simulating a move of M, M' first simulates the λ-moves of N until N waits for input (an instruction of F_M). In the meantime, M' has received from N the test result ρ' of the new configuration of M, and has stored ρ' in its finite control. This enables M' to simulate the next move of M. Note that, initially, M' starts by executing λ-moves of N (and already has the correct test result). From these observations it should be clear that $\tau(M') = \tau(M)$. ∎

It is quite obvious that this theorem also holds for many other automaton types Y not used in this paper (e.g., 1-way multi-head automata, multi-tape automata, etc.). However, it obviously does not hold for time-restricted automata; in that case the notion of simulation should be adapted, putting appropriate time restrictions on the $1\text{DT} - X$ transducers too (cf. Engelfriet and Hoogeboom, 1989).

1.3. *Pushdowns and Iterated Pushdowns*

The easiest way to define the storage type of an iterated pushdown is to view the pushdown as an operation on storage types, and to iterate this operation; see Greibach (1970), Engelfriet (1986), Engelfriet and Vogler (1986, 1987, 1988), and Vogler (1986). We will discuss several equivalent ways of defining this operation, and we will show that it preserves equivalence.

Let $X = (C, T, F, m, C_0, \mathrm{id})$ be a storage type. Let Γ be a fixed, infinite set of pushdown symbols. The storage type *pushdown of* X, denoted $P(X)$, has configurations that are pushdowns of which each square contains a pair (γ, c), where γ is a pushdown symbol and c an X-configuration. As usual, a $P(X)$ automaton M has access to the top-most square (γ, c) of its pushdown only. It can test which symbol γ is in that square, and it can apply the tests from T to c. Also as usual, M can change the pushdown by popping the top-most square or by pushing a new square. The push instruction contains the symbol of the new square, and contains an instruction from F that should be applied to c in order to obtain the X-configuration of the new square.

Formally, $P(X) = (C', T', F', m', C_0', \mathrm{id}')$, where

$$C' = (\Gamma \times C)^+ \text{ and } C_0' = \Gamma \times C_0,$$
$$T' = \{\mathrm{top} = \gamma \mid \gamma \in \Gamma\} \cup \{\mathrm{test}(t) \mid t \in T\},$$
$$F' = \{\mathrm{push}(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup \{\mathrm{pop}, \mathrm{id}'\},$$

and, for every $c' = \beta(\mu, c)$ with $\beta \in (\Gamma \times C)^*$, $\mu \in \Gamma$, and $c \in C$:

$m'(\mathrm{top} = \gamma)(c') = (\mu = \gamma)$,

$m'(\mathrm{test}(t))(c') = m(t)(c)$,

$m'(\mathrm{push}(\gamma, f))(c') = \beta(\mu, c)(\gamma, m(f)(c))$ if $m(f)$ is defined on c, and undefined otherwise,

$m'(\mathrm{pop})(c') = \beta$ if $\beta \neq \lambda$ and undefined otherwise, and

$m'(\mathrm{id}')(c') = c'$.

Remarks. (1) The top of the pushdown is to the right. For reasons of simplicity there is no empty pushdown.

(2) Whenever no confusion can arise, the prime is dropped from id'.

(3) A pushdown alphabet $\Gamma_M \subseteq \Gamma$ is specified for every $P(X)$ automaton M. In fact it is always possible to take Γ_M to contain the γ that occurs in the initial $P(X)$-configuration of M, and all γ's that occur in $\mathrm{push}(\gamma, f)$ instructions used by M (as determined by F'_M).

(4) For a pushdown $c' = (\gamma_0, c_0)(\gamma_1, c_1)\cdots(\gamma_n, c_n) \in (\Gamma \times C)^+$, we will call $\gamma_0\gamma_1\cdots\gamma_n$ the *symbol part* and $c_0c_1\cdots c_n$ the *X-configuration part* of c'.

The operation $P(X)$ on storage types can now be iterated: for a storage type X, $P^0(X) = X$, and, for $k \geqslant 1$, $P^{k+1}(X) = P(P^k(X))$. For $k \geqslant 0$, the *k-iterated pushdown* is the storage type $P^k(X_0)$, also denoted P^k. Any P^k automaton is called an *iterated pushdown automaton.* One-way iterated pushdown automata were considered in, e.g., Maslov (1976), Damm and Goerdt (1986), Engelfriet and Vogler (1987), and Vogler (1986). Note that the 1-iterated pushdown $P(X_0)$ is equivalent to the usual pushdown P (as defined in Section 1.1); i.e., $P(X_0) \equiv P$. In fact, $P(X_0)$ just has the additional X_0-configuration c_0 in each square of the pushdown, which has no influence on the computation of any $P(X_0)$ automaton.

It is possible to strengthen $P(X)$ by allowing an additional test "bottom" that is true for one-square pushdowns, and additional instructions "stay(γ, f)" with $\gamma \in \Gamma$ and $f \in F$ that transform $\beta(\mu, c)$ into $\beta(\gamma, m(f)(c))$. Let the resulting storage type be denoted by $P_s(X)$: *pushdown of X with stay instructions.* In fact, $P_s(X)$ is not really stronger than $P(X)$ because they are equivalent storage types: $P_s(X) \equiv P(X)$. Clearly $P(X) \leqslant P_s(X)$ is trivial. Let us show now that $P_s(X) \leqslant P(X)$. Let M be a 1-way deterministic $P_s(X)$ transducer. We have to show that M can be changed into an equivalent 1-way deterministic $P(X)$ transducer. It is easy for M to get rid of the bottom test: it just marks the bottom square symbol and keeps it marked. The stay(γ, f) instruction can be simulated by a push($\bar{\gamma}, f$) instruction, where the bar means that the square below this one is garbage; thus, each pop instruction should be replaced by a subroutine

while the top symbol is barred **do** pop;
pop.

In this way it should be clear that $P_s(X) \equiv P(X)$.

On the other hand $P(X)$ can be weakened, e.g., by taking two pushdown symbols instead of infinitely many, say 0 and 1. We will denote by $P_{\{0,1\}}(X)$ the storage type that is defined in exactly the same way as $P(X)$ except that $\{0, 1\}$ is used instead of Γ. It is well known that ordinary pushdown automata only need two pushdown symbols, and this also holds for $P(X)$ automata because $P_{\{0,1\}}(X) \equiv P(X)$. Again, $P_{\{0,1\}}(X) \leqslant P(X)$ is trivial. To see that $P(X) \leqslant P_{\{0,1\}}(X)$, let $\Gamma_M = \{\gamma_1, \gamma_2, ..., \gamma_n\}$, and simulate a pushdown square (γ_i, c) by the piece of pushdown $(1, c)(0, c)^i$. Then push(γ_i, f) is simulated by a push $(1, f)$ followed by i times push(0, id), and pop is simulated by **while** top $= 0$ **do** pop followed by one pop, and test(t) remains the same. Finally, the top symbol can be determined (in the finite control) by first executing **while** top $= 0$ **do** pop, meanwhile counting

the number i of iterations, and then executing i push(0, id) instructions to restore the old pushdown.

Thus we can feel at ease that several of the usual variations for pushdowns can also be used for $P(X)$. But what about iterated pushdowns? E.g., is $P_s^k \equiv P^k$? In other words, can we use stay instructions at all levels of the iterated pushdown? To show this we need the fact that the operation $P(X)$ preserves $\equiv$, or better, that $P(X)$ is monotonic with respect to $\leqslant$ (cf. Theorem 4.22 of Engelfriet and Vogler, 1986). Note that this is a very natural requirement for any operation on storage types.

THEOREM 1.3.1. *Let X_1 and X_2 be two storage types.*

If $X_1 \leqslant X_2$, then $P(X_1) \leqslant P(X_2)$.
If $X_1 \equiv X_2$, then $P(X_1) \equiv P(X_2)$.

Proof. The proof is similar to that of Theorem 1.2.4. Let $X_i = (C_i, T_i, F_i, m_i, C_{0i}, \mathrm{id}_i)$ for $i = 1, 2$, and assume that $X_1 \leqslant X_2$, i.e., $1\mathrm{DT} - X_1 \subseteq 1\mathrm{DT} - X_2$. We have to show that $1\mathrm{DT} - P(X_1) \subseteq 1\mathrm{DT} - P(X_2)$. Since, as discussed above, $P_s(X_2) \equiv P(X_2)$, it suffices to show that $1\mathrm{DT} - P(X_1) \subseteq 1\mathrm{DT} - P_s(X_2)$. Let M be a 1-way deterministic $P(X_1)$ transducer with initial configuration $(\gamma_0, c_0) \in \Gamma \times C_{01}$. Let T_M denote the set of all $t \in T_1$ that occur in the tests test(t) used by M, and let F_M denote the set of all $f \in F_1$ that occur in the instructions push(γ, f) used by M. As in the proof of Theorem 1.2.4, the use that M makes of the storage X_1 (through its $P(X_1)$ storage) is fully determined by the X_1-transduction $\tau = \tau(T_M, F_M, c_0)$. Let N be a 1-way deterministic X_2 transducer realizing τ. We will give an informal description of a 1-way deterministic $P_s(X_2)$ transducer M' that simulates M; the formal construction is left to the reader. M' has the same states, initial state, and final states as M (and, of course, the same input and output alphabets as M). M' simulates a pushdown square (γ, c) of M, with $\gamma \in \Gamma$ and $c \in C_1$, by a pushdown square $(\langle \gamma, p, \rho \rangle, d)$, where p is a state of N, ρ is the test result of c for T_M, and $d \in C_2$. This clearly allows M' to obtain the test result of (γ, c) for the $P(X_1)$ tests used by M. M' simulates a pop instruction of M by a pop instruction. Thus, it only remains to explain the simulation by M' of a push(γ, f) instruction of M. Consider a pushdown $(\gamma_0, c_0)(\gamma_1, c_1) \cdots (\gamma_n, c_n)$ arisen during a computation of M. Clearly, this pushdown was built up from the initial $P(X_1)$-configuration (γ_0, c_0) by applying instructions push(γ_1, f_1), ..., push(γ_n, f_n), in that order, such that $c_i = m_1(f_i)(c_{i-1})$ for $1 \leqslant i \leqslant n$. When simulating M, M' has a corresponding pushdown

$$(\langle \gamma_0, p_0, \rho_0 \rangle, d_0)(\langle \gamma_1, p_1, \rho_1 \rangle, d_1) \cdots (\langle \gamma_n, p_n, \rho_n \rangle, d_n),$$

where p_i and d_i are the state of N and the X_2-configuration of N, respectively, that are obtained when the string $f_1 \cdots f_i$ is fed into N (executing the maximal number of λ-moves of N); note that N then produces output $\rho_1 \cdots \rho_i$, and, as in the proof of Theorem 1.2.4, we may assume that ρ_i is not produced by N before it reads f_i. A push(γ, f) instruction of M is simulated by M' by first executing a push$(\langle \gamma, p, \rho \rangle, g)$ instruction where p and g are the new state of N and the instruction executed by N when feeding the symbol f into N (in state p_n and configuration d_n); in case N produces output during this move, ρ is this output, otherwise $\rho = \rho_n$. The simulation of push(γ, f) is then continued by simulating all following λ-moves of N, storing output of N (if it occurs) at the third position of the pushdown symbol; to do this M' executes instructions of the form stay$(\langle \gamma, p', \rho' \rangle, g')$.

Finally we note that the initial $P(X_2)$-configuration of M' is $(\langle \gamma_0, p_0', \rho_0 \rangle, d_0')$, where p_0' is the initial state of N, $\rho_0 = \rho(T_M, c_0)$, and d_0' is the initial X_2-configuration of N. M' should start its work by executing all λ-moves of N, thus changing $(\langle \gamma_0, p_0', \rho_0 \rangle, d_0')$ into $(\langle \gamma_0, p_0, \rho_0 \rangle, d_0)$. ∎

From this theorem it can easily be concluded, e.g., that $P_s^k(X_0) \equiv P^k(X_0)$, i.e., that stay instructions can be used at all levels of the k-iterated pushdown. The proof is by induction: the case $k = 0$ is trivial; assuming that $P_s^k(X_0) \equiv P^k(X_0)$, application of P to both sides is allowed by the previous theorem and gives $P(P_s^k(X_0)) \equiv P^{k+1}(X_0)$; using the equivalence $P(X) \equiv P_s(X)$ for $X = P_s^k(X_0)$ then gives $P(P_s^k(X_0)) \equiv P_s(P_s^k(X_0)) = P_s^{k+1}(X_0)$. This simple proof illustrates our technique of dealing with iterated pushdown automata, as mentioned in the introduction to this section. First we show that stay instructions can be used in all $P(X)$ automata, generalizing an obvious property of ordinary pushdown automata. Then we iterate the $P(X)$ operation, thus obtaining that stay instructions can be used in all P^k automata.

Let us draw two other easy conclusions from Theorem 1.3.1. First, we already argued that $P(X_0) \equiv P$. Repeated application of Theorem 1.3.1 now gives that $P^k(X_0) \equiv P^{k-1}(P)$ for every $k \geqslant 1$. Thus we may assume that the innermost pushdown squares of an iterated pushdown just contain pushdown symbols. Second, it is easy to see that, for every storage type X, $X \leqslant P(X)$. In fact, every instruction f can be simulated by a push(γ, f), and every test t by test(t). Thus, iterated application of Theorem 1.3.1 gives that $P^k \leqslant P^{k+1}$ for every $k \geqslant 0$. Hence, by Theorem 1.2.4, $Y - P^k \subseteq Y - P^{k+1}$ for every $k \geqslant 0$ and every automaton type Y.

Later we will consider other operations $U(X)$ on storage types. For such an operation we always define $U^0(X) = X$ and $U^{k+1}(X) = U(U^k(X))$, and we denote $U^k(X_0)$ by U^k. For every such U, it can be shown that $X \leqslant U(X)$

for every X, and that U is monotonic with respect to $\leqslant$. Hence, as above, $U^k \leqslant U^{k+1}$ for all k. We usually leave it to the reader to prove these facts; the proof of monotonicity is always similar to that of Theorem 1.3.1.

2. Iterated Pushdown Automata

In this section we show the results in Fig. 1, except for the nondeterministic r-head case. As remarked before, these results will be shown by induction on the number k of iterations of the pushdown operation. In all cases, the basis of the induction is obtained from Cook's well-known characterization of the nondeterministic auxiliary pushdown automata by time complexity classes.

Proposition 2.1 (Cook, 1971). *For* $s(n) \geqslant \log n$, $\mathrm{NSPACE}(s(n)) - P = \bigcup \mathrm{DTIME}(2^{ds(n)})$.

The induction step is obtained by "taking" two results of Ladner, Lipton, and Stockmeyer (1984) and Ruzzo (1980) on auxiliary pushdown automata, and generalizing them to auxiliary $P(X)$ automata in a straightforward way. Thus, in a certain sense, we obtain our results "for free." In the first of these two results (stated in Theorems 2.2 and 2.3) alternating auxiliary pushdown (of X) automata are considered, and it is shown that the pushdown is equivalent to exponentially more space.

Theorem 2.2. *For any storage type* X *and* $s(n) \geqslant \log n$, $\mathrm{ASPACE}(s(n)) - P(X) = \bigcup \mathrm{ASPACE}(2^{ds(n)}) - X$.

Proof. The proof is a rather straightforward generalization of the proof of Theorem 3.1 of (Ladner, Lipton, and Stockmeyer, 1984), where it is shown that $\mathrm{ASPACE}(s(n)) - P = \bigcup \mathrm{ASPACE}(2^{ds(n)})$, i.e., the result for $X = X_0$. Nevertheless we will discuss the proof here, so that it can be adapted to proofs of later theorems. Let $X = (C, T, F, m, C_0, \mathrm{id})$.

(i) We have to show that, for any $d > 0$, $\mathrm{ASPACE}(2^{ds(n)}) - X \subseteq \mathrm{ASPACE}(s(n)) - P(X)$. Let M be an alternating auxiliary $\mathrm{SPACE}(2^{ds(n)})$ X automaton. As in Ladner, Lipton, and Stockmeyer (1984) we may assume that $d \geqslant 1$ and that M has only one tape, i.e., a worktape and no input tape. Thus, an ID of M can be viewed as a pair (α, c) such that $c \in C$ and $\alpha \in \Phi^*(Q \times \Phi)\Phi^*$ with $|\alpha| = 2^{ds(n)}$, where Φ is the worktape alphabet and Q the set of states of M. As usual, in α, the pair $(q, \phi) \in Q \times \Phi$ indicates that M is in state q scanning ϕ. Moreover we may assume that each ID of M has at most two successors. Thus the transition function of M is a partial function $\delta\colon Q \times \Phi \times R(T_M) \to (Q \times \{-1, 0, +1\} \times \Phi \times F_M)^2$, where T_M and

F_M are the tests and instructions used by M to manipulate its X-storage. This is just as in Ladner, Lipton, and Stockmeyer (1984), with $R(T_M)$ and F_M added.

The ASPACE($s(n)$) $-$ $P(X)$ automaton M' that simulates M, starts by laying off a block of $s(n)$ squares on its worktape (assuming that $s(n)$ is space constructable; otherwise M' just guesses $s(n)$ nondeterministically). In general, M' simulates M by storing a computation $(\alpha_0, c_0) \vdash (\alpha_1, c_1) \vdash \cdots \vdash (\alpha_k, c_k)$ of M on its pushdown, where (α_k, c_k) is the current ID of M (note that a computation is a path in a computation tree). To be more precise, the symbol part of the pushdown of M' contains the string $\alpha_0 m_1 \alpha_1 m_2 \alpha_2 \cdots m_k \alpha_k$, where $m_i \in \{1, 2\}$ indicates whether (α_i, c_i) is the first or second successor of (α_{i-1}, c_{i-1}). Furthermore, each square of α_0 contains c_0, and each square of $m_i \alpha_i$ contains c_i, $1 \leqslant i \leqslant k$. Thus the X-configuration part of the pushdown contains $c_0, c_1, \ldots, c_k$ with c_i appearing $|m_i \alpha_i|$ times. Note that as soon as c_i appears (by some push(m_i, f) instruction), it can be duplicated for all squares of α_i by appropriate push(γ, id) instructions. Moreover, M' keeps the element of $Q \times \Phi$ of the topmost ID (α_k, c_k) in its finite control. Since the test result of the pushdown includes the test result of c_k for T_M (due to the tests of the form test(t), $t \in T_M$, of M'), we may in fact assume that M' keeps the element of $Q \times \Phi \times R(T_M)$ in its finite control that corresponds to the current ID (α_k, c_k) of M, i.e., an element x of the domain of the transition function δ of M. A next move of M is now simulated by M' as follows. First the symbol $m_{k+1} \in \{1, 2\}$ is chosen universally (existentially) by M' if (α_k, c_k) is a universal (existential, respectively) ID of M. Now, x and m_{k+1} together determine the next move of M, including an instruction $f \in F_M$. Thus, M' executes the push(m_{k+1}, f) instruction, and then nondeterministically (i.e., existentially) pushes the symbols of a new α_{k+1}, one by one, executing push(γ, id) instructions. M' can count to $2^{ds(n)}$ (the length of α_{k+1}) using its $s(n)$-bounded worktape. M' ensures that (α_{k+1}, c_{k+1}) is the correct successor of (α_k, c_k), according to x and m_{k+1}, by a universal branch after each push of a symbol of α_{k+1}. In this universal branch it compares this symbol with the corresponding one(s) in α_k by popping (roughly) $2^{ds(n)}$ symbols, again using its worktape as a counter. Note that, as usual, the jth symbol of α_{k+1} is determined by x, m_{k+1}, and the $(j-1)$th, jth, and $(j+1)$th symbols of α_k.

Initially M' installs the initial ID (α_0, c_0) of M by copying the input string (of length n) to the pushdown, extending it with blanks to length $2^{ds(n)}$, and duplicating the initial X-configuration c_0 of M (M' has an initial $P(X)$-configuration (γ, c_0) for some γ). M' accepts whenever (α_k, c_k) is an accepting ID of M.

For more details see Ladner, Lipton, and Stockmeyer (1984). Note that in fact the m_i are superfluous (M' only needs m_{k+1}, which it may as well

keep in its finite control). However, they are useful in later variations of this proof.

(ii) We have to show that $\mathrm{ASPACE}(s(n)) - P(X) \subseteq \bigcup \mathrm{ASPACE}(2^{ds(n)}) - X$. In this direction the proof is based on ideas of Cook, in his proof of Proposition 2.1. In Ladner, Lipton, and Stockmeyer (1984) it is unfortunately shown in this direction that $\mathrm{ASPACE}(s(n)) - P \subseteq \bigcup \mathrm{DTIME}(\exp_2(ds(n)))$, which equals $\bigcup \mathrm{ASPACE}(2^{ds(n)})$ by a well-known result of Chandra, Kozen, and Stockmeyer (1981). However, the largest and most important part of that proof can be taken over.

Let M be an alternating auxiliary $\mathrm{SPACE}(s(n))$ $P(X)$ automaton. As in Ladner, Lipton, and Stockmeyer (1984), we may assume that M behaves deterministically while it is either pushing or popping; in other words, if, for the transition function δ of M, $\delta(a, \rho)$ contains more than one element, then all elements of $\delta(a, \rho)$ contains more than one element, then all elements of $\delta(a, \rho)$ are of the form (b, id), where id is the identity instruction of $P(X)$. As a consequence (as in Ladner, Lipton, and Stockmeyer, 1984), each ID of M may be thought of as being in one of three possible modes: PUSH, POP, or IDLE (depending on whether the appropriate $\delta(a, \rho)$ contains push(γ, f), pop, or id instructions, respectively). The IDLE ID's are partitioned into U-IDLE and E-IDLE ID's, depending on whether they are universal or existential, respectively.

The notion of a ($s(n)$-bounded) surface ID can be defined as in Ladner, Lipton, and Stockmeyer (1984); it now also involves an X-configuration. Thus, a *surface ID* is an ID $(\alpha, (\gamma, c))$ of M, where (γ, c) is a one-square $P(X)$-configuration, and α is the rest of the ID, containing a worktape of length $s(n)$. Define $\mathrm{top}(\alpha, (\gamma, c)) = (\gamma, c)$. Intuitively (γ, c) is the top square of a pushdown in an ordinary ID: every ID of M has an associated surface ID, obtained by replacing the pushdown by its top square. For an ID w we denote the associated surface ID by $\mathrm{surf}(w)$. It would now be possible to define a surface computation just as in Ladner, Lipton, and Stockmeyer (1984), where the top square now plays the role of the top symbol in Ladner, Lipton and Stockmeyer (1984). For our purposes it suffices to use the following simplified notion of surface computation: a *surface computation* is a computation tree of M of which both the root and the leaves are labeled by ID's with one-square pushdowns, i.e., by surface ID's. Obviously, the root ID and the leaf ID's have the same pushdown square (because there are no stay(γ, f) instructions in $P(X)$), and in particular the same X-configuration. For surface ID's $r, z_1, \ldots, z_k$ we write $r \to \{z_1, \ldots, z_k\}$ if there is a surface computation whose root is labeled r and whose leaf labels are contained in the set $\{z_1, \ldots, z_k\}$; thus we may assume that $\mathrm{top}(r) = \mathrm{top}(z_i)$ for all $1 \leqslant i \leqslant k$.

For every input string x there is an initial (surface) ID $I(x) = (\alpha_0, (\gamma_0, c_0))$ of M, where α_0 contains x. Clearly we may also assume that

there is exactly one accepting (surface) ID with input string x, say, $A(x) = (\beta_0, (\gamma_0, c_0))$, where β_0 differs only from α_0 in its state; thus, M accepts by (unique) final state, one-square pushdown, and blank worktape. Consequently, x is accepted by M iff $I(x) \to \{A(x)\}$.

In Ladner, Lipton, and Stockmeyer (1984) a "proof system," consisting of six "proof rules," is given in which "terms" of the form $r \to \{z_1, ..., z_k\}$ can be "proven," where $r, z_1, ..., z_k$ are surface ID's. Exactly the same proof system can be used here, except that we restrict all terms $r \to \{z_1, ..., z_k\}$ to those that satisfy $\text{top}(r) = \text{top}(z_i)$ for all $1 \leqslant i \leqslant k$. This restriction could also have been made in Ladner, Lipton, and Stockmeyer (1984), where it is only mentioned in proof rule 4. For completeness's sake we list the proof rules below:

1. If true, then $r \to \{r\}$.
2. (a) If $r \to W$ and $W \subseteq V$, then $r \to V$.
   (b) If $r \to W \cup \{w\}$ and $w \to V$, then $r \to W \cup V$.
3. (a) If r is in $E-$IDLE mode and $r \vdash w$, then $r \to \{w\}$.
   (b) If r is in U-IDLE mode and $\{w_1, ..., w_k\} = \{w \mid r \vdash w\}$, then $r \to \{w_1, ..., w_k\}$.
4. If r is in PUSH mode,

   $r \vdash w'$,
   $w = \text{surf}(w')$,
   $w \to \{v_1, ..., v_k\}$,
   for every $1 \leqslant i \leqslant k$
   v_i is in POP mode,
   $v_i' \vdash z_i$,
   $\text{surf}(v_i') = v_i$,

   then $r \to \{z_1, ..., z_k\}$.

In these rules, lower case letters stand for surface ID's, primed lower case letters stand for ID's with two-square pushdowns, and upper case letters stand for finite sets of surface ID's. Note that each proof rule consists of zero, one, or two antecedents (between "if" and "then") and one consequent (after "then"), together with some application conditions (also between "if" and "then"). As an example, rule 2(a) has antecedent $r \to W$, application condition $W \subseteq V$, and consequent $r \to V$. In Lemma 3.2 of Ladner, Lipton, and Stockmeyer (1984) it is shown that the proof system is sound and complete, i.e., that $r \to \{z_1, ..., z_k\}$ can be proven iff it is true. The proof of that Lemma 3.2 stays valid here (in Ladner, Lipton, and Stockmeyer, 1984, a more involved notion of surface computation is used,

to facilitate the proof of Lemma 3.2; this notion can easily be adapted to our case).

It remains to show that there is an alternating auxiliary SPACE($2^{ds(n)}$) X automaton M', for some $d>0$, that checks whether $I(x) \to \{A(x)\}$ for a given input x. To do this, M' guesses a "proof tree" for $I(x) \to \{A(x)\}$. As usual, such a proof tree is a tree whose nodes are labeled by terms $r \to \{z_1, ..., z_k\}$ and whose root is labeled by $I(x) \to \{A(x)\}$. Moreover, for each nonleaf node, its label and the labels of its children are the consequent and the antecedents (respectively) of an instance of a proof rule. M' guesses the proof tree top-down, using the proof rules in a backward fashion, in such a way that its computation tree mirrors the proof tree. More precisely, at each moment of its computation the current ID of M' contains a coded version of a term $r \to \{z_1, ..., z_k\}$. M' chooses existentially an instance of a rule whose consequent is $r \to \{z_1, ..., z_k\}$, and then branches universally to check each of the antecedents of the rule; it accepts in case there are no antecedents (proof rules 1 and 3). Note that actually M' branches universally in proof rule 2(b) only.

M' stores a term $r \to \{z_1, ..., z_k\}$ by keeping the X-configuration of r, $z_1, ..., z_k$ as its own X-configuration (recall that $\text{top}(r) = \text{top}(z_i)$ for all i), and keeping everything else on its worktape. Since "everything else" mainly consists of the worktape contents of r, $z_1, ..., z_k$ (which are of length $s(n)$), it is easy to see that M' only needs workspace $2^{ds(n)}$ for some d (note that $z_1, ..., z_k$ are all different). Clearly this workspace also suffices to construct an antecedent out of a consequent (and to test the application conditions). Note that, in the construction of such an antecedent, if M tests its pushdown (as needed to obtain an instance of proof rule 3 or 4), M' can test the corresponding top square; in particular, a test test(t) of M is simulated by the test t of M'. Furthermore, if M executes a push(γ, f) instruction (as needed for an instance of proof rule 4), M' just applies f; this results in the X-configuration of the surface ID's w, $v_1, ..., v_k$ of the antecedent $w \to \{v_1, ..., v_k\}$. Note that, when using proof rule 4, M' does not have to construct the ID's $w', v_1', ..., v_k'$ explicitly. Initially, M' stores the term $I(x) \to \{A(x)\}$, with $I(x) = (\alpha_0, (\gamma_0, c_0))$ and $A(x) = (\beta_0, (\gamma_0, c_0))$ where (γ_0, c_0) is the initial configuration of M. Thus M' has initial configuration c_0. This ends the description of M'. ▮

Since it is easy to see that $2\text{A(multi)} - X = \text{ASPACE}(\log n) - X$, generalizing the well-known equivalence of multi-head automata and SPACE($\log n$) Turing machines, it follows from Theorem 2.2 that $2\text{A(multi)} - P(X) = \text{ASPACE(poly)} - X$. In the next theorem we show that the number of heads of the multi-head automaton corresponds to the exponent of the polynomial. Moreover, in the case of one head, it may be restricted to be one-way.

THEOREM 2.3. *For any storage type* X,

(1) *for every* $r \geqslant 1$, $2A(r) - P(X) = ASPACE(n^r) - X$, *and*

(2) $1A - P(X) = 2A(1) - P(X) = ASPACE(n) - X$.

Proof. (i) The proof that $ASPACE(n^r) - X \subseteq 2A(r) - P(X)$ is exactly the same as in part (i) of the proof of Theorem 2.2. This is because r heads can be used to count to n^r.

In case $r = 1$, the head only needs to move one way to count to n. However, since the head cannot be reset, care should be taken that counting is done only in universal side branches. Thus, the fact that (α_{k+1}, c_{k+1}) is of the correct length should be checked after guessing α_{k+1}. Similarly, α_0 should first be guessed and then compared to the input; to permit this, the symbol part of the pushdown should contain $\alpha_0^R m_1 \alpha_1^R \cdots m_k \alpha_k^R$ rather than $\alpha_0 m_1 \alpha_1 \cdots m_k \alpha_k$, where α^R is the reverse of α. For details see the proof of Theorem 5.4 of Chandra, Kozen, and Stockmeyer (1981), where it is shown that $ASPACE(n) \subseteq 1A - P$.

(ii) The proof that $2A(r) - P(X) \subseteq ASPACE(n^r) - X$ is the same as in part (ii) of the proof of Theorem 2.2. The only difference is in the construction of the $ASPACE(n^r) - X$ automaton M' that simulates the $2A(r) - P(X)$ automaton M. M' should be able to store a term $z_0 \to \{z_1, ..., z_k\}$, where z_0, $z_1, ..., z_k$ are surface ID's of M. The X-configuration of $z_0, z_1, ..., z_k$ is no problem. For fixed X-configuration, M has $O(n^r)$ possible surface ID's. Since they are of size $O(\log n)$, not counting the input, M' does not have enough worktape to store them all. However, they can be generated by M' in a systematic order. Thus, to store $z_0 \to \{z_1, ..., z_k\}$, M' keeps z_0 directly on its worktape, and for $z_1, ..., z_k$ it keeps a boolean array A of length $O(n^r)$ on its worktape, where $A[i]$ indicates whether or not the ith surface ID, in the above order, is in the set $\{z_1, ..., z_k\}$. Whenever M' needs the ith surface ID, M' can generate it and keep it on its worktape. In this way M' can compute antecedents from consequents (and test application conditions), as required by the proof rules. ∎

In the second result that is needed for our induction step, we show the equivalence of alternation and an auxiliary pushdown (Theorem 2.4). This equivalence was first established in Chandra, Kozen, and Stockmeyer (1981), where it is shown that $ASPACE(s(n)) = \bigcup DTIME(2^{ds(n)})$ and hence that $ASPACE(s(n)) = NSPACE(s(n)) - P$, by Cook's result (Proposition 2.1). In Theorem 1 of Ruzzo (1980) a direct proof of the latter equality is given, in order to show that time on the pushdown automaton corresponds to (computation) tree size on the alternating automaton. In this way Ruzzo (1980) strengthens the equivalence alternation ≡ pushdown. Here we strengthen it by generalizing it to X automata.

THEOREM 2.4. *For any storage type X and $s(n) \geqslant \log n$,* $\mathrm{NSPACE}(s(n)) - P(X) = \mathrm{ASPACE}(s(n)) - X$.

Proof. (i) We have to show that $\mathrm{ASPACE}(s(n)) - X \subseteq \mathrm{NSPACE}(s(n)) - P(X)$. This could be done as in the proof of Theorem 1 of Ruzzo (1980). Here we adapt part (i) of the proof of Theorem 2.2; compared to that proof, when simulating an $\mathrm{ASPACE}(s(n)) - X$ automaton by an auxiliary $P(X)$ automaton, we have space $s(n)$ rather than space $\log(s(n))$, but we do not have alternation. The construction of M' from M is the same as in Theorem 2.2 except for two differences, to get rid of universal branching.

First, to ensure that (α_{k+1}, c_{k+1}) is the correct successor of (α_k, c_k), α_{k+1} is computed from α_k on the worktape of M' (where there is enough space now).

Second, M' does not simulate a universal move of M by a corresponding universal move, but executes the two universal branches one after another. Thus, rather than mirroring the computation tree of M, M' does "a simple depth first search of M's computation tree, using its stack to backtrack through the universal nodes" (Ruzzo, 1980). More precisely, if the top ID (α_k, c_k) is a universal ID of M, then M' pushes $m_{k+1} = 1$ and pushes the first successor of (α_k, c_k). When the top ID is an accepting ID of M, M' pops ID's until the top ID is a universal ID (α_k, c_k) with $m_{k+1} = 1$. It then pushes $m_{k+1} = 2$ and pushes the second successor of (α_k, c_k). Note that to find out that (α_k, c_k) is universal, and to compute its second successor, M' can copy α_k to its worktape. M' accepts if it reaches the bottom of its pushdown.

This second change could also have been used in part (i) of the proof of Theorem 2.2. In other words, in that proof, alternation was needed only to check that (α_{k+1}, c_{k+1}) is a successor of (α_k, c_k).

(ii) We have to show that $\mathrm{NSPACE}(s(n)) - P(X) \subseteq \mathrm{ASPACE}(s(n)) - X$. The proof is exactly the same as part (ii) of the proof of Theorem 2.2. The only difference is that all computation trees and hence all surface computations consist of one path only, and thus have one leaf only. This means that the proof system can be restricted to terms $r \to \{z\}$, i.e., to pairs of surface ID's, just as in Cook's original proof of Proposition 2.1 (see also the second part of the proof of Theorem 1 of Ruzzo, 1980). Clearly such terms can be stored in space $O(s(n))$ rather than $O(2^{ds(n)})$. ∎

These generalizations now allow us to prove the main results of this section: all results of Fig. 1, except for the $2\mathrm{N}(r) - P^k$ automata. The proofs are by straightforward inductions.

THEOREM 2.5. *For any $k \geqslant 1$ and $s(n) \geqslant \log n$,* $\mathrm{NSPACE}(s(n)) - P^k = \mathrm{ASPACE}(s(n)) - P^{k-1} = \bigcup \mathrm{DTIME}(\exp_k(ds(n)))$.

Proof. Since the first equality is just Theorem 2.4, for $X = P^{k-1}$, it remains to show that this class equals $\bigcup \mathrm{DTIME}(\exp_k(ds(n)))$. The proof is by induction on k. For $k = 1$ this is Cook's theorem (Proposition 2.1), and the induction step is immediate from Theorem 2.2. ∎

THEOREM 2.6. *For any* $k \geq 1$, $\mathrm{2N(multi)} - P^k = \mathrm{2A(multi)} - P^{k-1} = \mathrm{DTIME}(\exp_{k-1}(\mathrm{poly}))$.

Proof. This is just Theorem 2.5 for $s(n) = \log n$, using the obvious facts that $\mathrm{2N(multi)} - X = \mathrm{NSPACE}(\log n) - X$ and $\mathrm{2A(multi)} - X = \mathrm{ASPACE}(\log n) - X$. ∎

It is shown in Kowalczyk, Niwinski, and Tiuryn (1989), by a more careful analysis, that $\mathrm{DSPACE}(s(n)) - P^k = \mathrm{NSPACE}(s(n)) - P^k$ (for space constructable $s(n)$), and thus $\mathrm{2D(multi)} - P^k = \mathrm{2N(multi)} - P^k$.

The reader who is interested in the application of Theorem 2.6 to one-way automata can jump directly to Section 7 and read Section 7.1 up to Theorem 7.5 and Section 7.2 up to Theorem 7.14.

THEOREM 2.7. *For any* $k \geq 1$,

(1) *for every* $r \geq 1$, $\mathrm{2A}(r) - P^k = \bigcup \mathrm{DTIME}(\exp_k(dn^r))$, *and*

(2) $\mathrm{1A} - P^k = \bigcup \mathrm{DTIME}(\exp_k(dn))$.

Proof. (1) By Theorem 2.3(1), for $X = P^{k-1}$, $\mathrm{2A}(r) - P^k = \mathrm{ASPACE}(n^r) - P^{k-1}$. By Theorem 2.5, for $s(n) = n^r$, this equals $\bigcup \mathrm{DTIME}(\exp_k(dn^r))$.

(2) By Theorem 2.3(2), for $X = P^{k-1}$, $\mathrm{1A} - P^k = 2A(1) - P^k$. ∎

In the remainder of this section we prove one half of the characterization of the $\mathrm{2N}(r) - P^k$ automata, viz. the inclusion $\mathrm{2N}(r) - P^k \subseteq \bigcup \mathrm{DTIME}(\exp_{k-1}(dn^{2r}))$ for $k \geq 2$. This result would follow from Theorem 2.7(1) if we could prove that $\mathrm{2N}(r) - P(X) \subseteq \mathrm{2A}(2r) - X$. One could believe this inclusion to be true by looking at the proof of $\mathrm{NSPACE}(s(n)) - P(X) \subseteq \mathrm{ASPACE}(s(n)) - X$ in Theorem 2.4. Since only pairs of surface ID's have to be stored by the new automaton, twice as many heads as the original automaton suffice. Unfortunately this does not suffice to simulate proof rule 2(b) backwards: a term $u \to \{v\}$ has to be replaced (universally) by two terms $u \to \{w\}$ and $w \to \{v\}$. Thus, the alternating automaton has to guess an arbitrary new surface ID w and use this in both universal branches. For this it would need $3r$ heads. To solve this problem we extend the power of $\mathrm{2A}(r) - X$ automata in an ad hoc fashion, calling them $\mathrm{2A}^+(r) - X$ automata.

A $\mathrm{2A}^+(r) - X$ automaton is a $\mathrm{2A}(r) - X$ automaton with the following additional features. First, any head may nondeterministically jump to an arbitrary position on the input tape. To this end the set $\{-1, 0, +1\}$ in the

transition function is replaced by $\{-1, 0, +1, \text{jump}\}$. Clearly, this feature alone would not strengthen the power of $2A(r)-X$ automata. Second, in a universal move of the automaton with, say, two universal branches, a finite number of relationships between the jumping heads in the first branch and those in the second branch may be specified. Such a relationship is of the form $\langle i, 1\rangle = \langle j, 2\rangle$, meaning that only those pairs of successors of the current ID are considered in which head i in the first branch is on the same position as head j in the second branch. This relationship can be required only if both i and j are jumping heads. We leave the formalization of this type of automaton to the reader.

First we show that this extension does not increase the power of alternating multi-head pushdown automata.

LEMMA 2.8. *For any storage type* X *and* $r \geqslant 1$, $2A^+(r)-P(X) = 2A(r)-P(X)$.

Proof. Let M be a $2A^+(r)-P(X)$ automaton. A $2A(r)-P(X)$ automaton M' can simulate M as follows. Just before a universal move with, say, two branches, M' guesses nondeterministically the new positions of the jumping heads in each branch, by pushing them in unary notation on the pushdown, using appropriate push(γ, id) instructions. Then M' simulates the universal move of M, without executing the pushdown instruction involved. Next, M' resets the jumping heads of M to the left endmarker $\not\in$ and puts them on their correct position by popping the pushdown. Finally, with the pushdown back in its configuration before the move, M' executes the pushdown instruction of M. ∎

Next we show that the extension solves the above-mentioned problem.

LEMMA 2.9. *For any storage type* X *and* $r \geqslant 1$, $2N(r)-P(X) \subseteq 2A^+(2r)-X$.

Proof. The proof is the same as in part (ii) of the proofs of Theorems 2.2 and 2.4. The new automaton M' only has to store terms $u \to \{v\}$, i.e., two surface ID's of the old automaton M. Clearly M' can do this with $2r$ heads, say, heads 1 to r for u, and heads $r+1$ to $2r$ for v. When simulating proof rule 2(b) backwards, $u \to \{v\}$ has to be replaced universally by $u \to \{w\}$ and $w \to \{v\}$ for an arbitrary surface ID w. Thus, for the first branch M' jumps with heads $r+1$ up to $2r$, and for the second branch with heads 1 up to r. The relationships between the jumping heads in both branches are $\langle r+i, 1\rangle = \langle i, 2\rangle$ for all $1 \leqslant i \leqslant r$; this ensures that heads $r+1$ to $2r$ in the first branch guess the same surface ID w as heads 1 to r in the second branch. ∎

These two lemmas together give us the following partial characterization.

LEMMA 2.10. *For any storage type X and $r \geq 1$, $2N(r) - P(P(X)) \subseteq ASPACE(n^{2r}) - X$.*

Proof. $2N(r) - P(P(X)) \subseteq 2A^+(2r) - P(X)$ by Lemma 2.9. The latter class equals $2A(2r) - P(X)$ by Lemma 2.8, which equals $ASPACE(n^{2r}) - X$ by Theorem 2.3(1). ∎

In the next section we will show that the inclusion in this lemma is in fact an equality (Theorem 3.2). Note that, taking $X = P^{k-2}$, Lemma 2.10 together with Theorem 2.5 show that $2N(r) - P^k \subseteq \bigcup DTIME(\exp_{k-1}(dn^{2r}))$, for $k \geq 2$.

3. ITERATED STACK AUTOMATA

The results of this section can be stated in one sentence: for 2-way multihead automata and for auxiliary $SPACE(s(n))$ automata, a stack has the same power as a pushdown of pushdowns. This implies that for iterated stack automata all results in Fig. 1 hold with k replaced by $2k$.

A stack automaton is a pushdown automaton with the additional ability of reading in the stack (Ginsburg, Greibach, and Harrison, 1967; Hopcroft and Ullman, 1979). Thus the storage type *stack of X*, denoted $SA(X)$, can be defined similarly to $P(X)$, with two additional instructions for moving its stack pointer up and down the stack: move-up and move-down, respectively. The reading tests of $SA(X)$ are called $\text{sym} = \gamma$ (for $\gamma \in \Gamma$) rather than $\text{top} = \gamma$, and the elements of Γ are now called stack symbols. A storage configuration of $SA(X)$ consists of a sequence $(\gamma_0, c_0)(\gamma_1, c_1) \cdots (\gamma_n, c_n)$ of "squares" (just as in $P(X)$), together with a number i $(0 \leq i \leq n)$ that indicates the position of the stack pointer (or stack head). If $i = n$, the configuration is said to be in *pushdown mode* (because only then the push and pop instructions are defined). If $i < n$, the configuration is said to be in *reading mode*. The move-down instruction decreases i by 1, and the move-up instruction increases i by 1. The test $\text{sym} = \gamma$ tests whether $\gamma_i = \gamma$, and the test $\text{test}(t)$ with $t \in T$ tests t on c_i. We leave it to the reader to give a more formal definition of $SA(X)$. It should also be clear that there exist several equivalent variations of $SA(X)$, where "equivalent" is meant in the formal sense of Section 1.2 (cf. the variations of $P(X)$ in Section 1.3). Thus, one could add tests "bottom" and "top" and instructions "stay(γ, f)" with their obvious meanings. One could also restrict the set of stack symbols to $\{0, 1\}$. As a final example, one could require $\text{test}(t)$ always to be false in reading mode: just after pushing, the test result of the new X-configuration can be stored in the symbol part of the new square (by an appropriate stay(γ, id) instruction). To guarantee that these variations can also be used

when $SA(X)$ is iterated, it should be proved that $SA(X)$ is monotonic with respect to $\leqslant$. This can be shown in exactly the same way as Theorem 1.3.1. Note finally that, trivially, $P(X) \leqslant SA(X)$.

The operation SA on storage types can be iterated just as P. For $k \geqslant 0$, *k-iterated stack* is the storage type $SA^k(X_0)$, also denoted SA^k. Any SA^k automaton is called an *iterated stack automaton.*

As remarked before, for multi-head and auxiliary automata, a stack has the same power as a pushdown of pushdowns. In fact, in one direction, a stack can be simulated by a pushdown of pushdowns.

LEMMA 3.1. *For every storage type* X, $SA(X) \leqslant P(P(X))$.

Proof. Let $X = (C, T, F, m, C_0, \mathrm{id})$. We have to show that $1DT - SA(X) \subseteq 1DT - P(P(X))$. It suffices to show that $1DT - SA(X) \subseteq 1DT - P_s(P(X))$, cf. Section 1.3 ($P_s$ is P with stay instructions). Let M be a 1-way deterministic $SA(X)$ transducer. An equivalent 1-way deterministic $P_s(P(X))$ transducer M' can be constructed as follows. The $SA(X)$-configuration of M in Fig. 2(a) is simulated by the $P(P(X))$-configuration of M' shown in Fig. 2(b). In Fig. 2, each blank square represents a pair (γ, c) with $\gamma \in \Gamma$ and $c \in C$. Each square with p or r, together with the "inner" pushdown above it, is a square of the "outer" pushdown in Fig. 2(b) (where the outer pushdown grows horizontally, and each inner pushdown grows vertically). The symbols p and r (with p, $r \in \Gamma$ but $\notin \Gamma_M$) stand for "pushdown mode" and "reading mode," respectively. Each inner pushdown is a prefix of the inner pushdowns to the left of it, and the number of inner pushdowns equals the number of squares in Fig. 2(a) that are above the

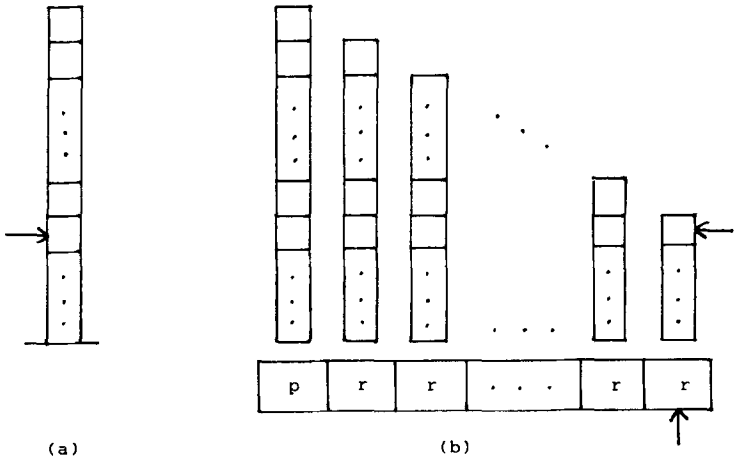

FIG. 2. Simulation of a stack by a pushdown of pushdowns.

stack pointer (including the square pointed at). The left-most inner pushdown (above p) has the same contents as the stack in Fig. 2(a).

The simulation of M by M' is easy. A move-down instruction of M is simulated by a push(r, pop) instruction of M'. This instruction adds another step to the "staircase' of Fig. 2(b). A move-up instruction of M is simulated by a pop instruction of M'. A push(γ, f) instruction, with $\gamma \in \Gamma$ and $f \in F$, is simulated by stay(p, push(γ, f)) provided top = p (and undefined otherwise). Similarly, a pop is simulated by stay(p, pop) provided top = p. A test sym = γ, with $\gamma \in \Gamma$, of M is simulated by the test test(top = γ) of M', and test(t), with $t \in T$, is simulated by test(test(t)). Of course, M' simulates the state behaviour and the input and output of M in the obvious way. Finally, if (γ_0, c_0) is the initial configuration of M, then (p, (γ_0, c_0)) is the initial configuration of M'. ▮

In general, a pushdown of pushdowns cannot be simulated by a stack (see next section). We now show that, for auxiliary and multi-head automata, the $P^2(X)$ automata can be simulated by the corresponding SA(X) automata. First we discuss the nondeterministic case, including the missing part of the nondeterministic r-head pushdown automata.

THEOREM 3.2. *For any storage type X, $s(n) \geqslant \log n$, and $r \geqslant 1$,*

(1) $\mathrm{NSPACE}(s(n)) - \mathrm{SA}(X) = \mathrm{NSPACE}(s(n)) - P(P(X))$
$= \bigcup \mathrm{ASPACE}(2^{ds(n)}) - X$, *and*

(2) $2\mathrm{N}(r) - \mathrm{SA}(X) = 2\mathrm{N}(r) - P(P(X)) = \mathrm{ASPACE}(n^{2r}) - X$.

Proof. (1) By Lemma 3.1 and Theorem 1.2.4, $\mathrm{NSPACE}(s(n)) - \mathrm{SA}(X) \subseteq \mathrm{NSPACE}(s(n)) - P(P(X))$. By Theorems 2.4 and 2.2, $\mathrm{NSPACE}(s(n)) - P(P(X)) = \bigcup \mathrm{ASPACE}(2^{ds(n)}) - X$. Thus, it remains to show that, for every $d > 0$, $\mathrm{ASPACE}(2^{ds(n)}) - X \subseteq \mathrm{NSPACE}(s(n)) - \mathrm{SA}(X)$. This is shown again by adapting part (i) of the proof of Theorem 2.2. Universal branching is avoided by the following two changes.

First, to ensure that (α_{k+1}, c_{k+1}) is the correct successor of (α_k, c_k), the stack automaton M' goes into reading mode each time a new symbol γ of α_{k+1} is pushed. It moves down to the corresponding symbols in α_k (using space $s(n)$ as a counter), compares those symbols to γ, and then moves up to the top again, to continue pushing α_{k+1}. This is the only use M' makes of its reading facility. Note that M' may as well first guess α_{k+1} completely, and then compare it to α_k, symbol for symbol. Second, M' backtracks through the computation tree of M, as explained in part (i) of the proof of Theorem 2.4.

(2) By Lemma 3.1 and Theorem 1.2.4, $2\mathrm{N}(r) - \mathrm{SA}(X) \subseteq 2\mathrm{N}(r) - P(P(X))$. By Lemma 2.10, $2\mathrm{N}(r) - P(P(X)) \subseteq \mathrm{ASPACE}(n^{2r}) - X$. Thus it remains to show that $\mathrm{ASPACE}(n^{2r}) - X \subseteq 2\mathrm{N}(r) - \mathrm{SA}(X)$. This can be

proven as in (1) above, where the only problem is how to ensure that (α_{k+1}, c_{k+1}) is the correct successor of (α_k, c_k) using r heads only (it would seem that, as in part (i) of the proof of Theorem 2.3, $2r$ heads are needed to count to n^{2r}). In Theorem 5 of (Hopcroft and Ullman, 1967b) it is explained how a 2-way one-head stack automaton can compare strings of length n^2 (see also Lemma 3 of Cook, 1971). The basic idea is to divide such strings into n blocks of length n each. In exactly the same way r heads can compare strings of length n^{2r}, by dividing them into n^r blocks of length n^r each (and using the heads to count to n^r). Obviously this method also works for SA(X) automata. ▮

With the next result the proof of Fig. 1 (for iterated pushdown automata) is completed.

THEOREM 3.3. *For any* $k \geq 2$ *and* $r \geq 1$, $2\mathrm{N}(r) - P^k = \bigcup \mathrm{DTIME}(\exp_{k-1}(dn^{2r}))$.

Proof. Both classes equal $\mathrm{ASPACE}(n^{2r}) - P^{k-2}$, by Theorems 3.2(2) and 2.5, respectively. ▮

Next we discuss alternating stack automata, including alternating *nonerasing stack automata.* A stack automaton is nonerasing if it does not use its pop instruction. The corresponding storage type NESA(X) is obtained from SA(X) by simply dropping the pop instruction from the set of instructions. Thus, trivially, $\mathrm{NESA}(X) \leqslant \mathrm{SA}(X)$. Also, NESA is monotonic with respect to $\leqslant$.

THEOREM 3.4. *For any storage type* X, $s(n) \geq \log n$, *and* $r \geq 1$,

(1) $\mathrm{ASPACE}(s(n)) - \mathrm{NESA}(X) = \mathrm{ASPACE}(s(n)) - \mathrm{SA}(X)$
$= \mathrm{ASPACE}(s(n)) - P(P(X)) = \bigcup \mathrm{ASPACE}(\exp_2(ds(n))) - X$,

(2) $2\mathrm{A}(r) - \mathrm{NESA}(X) = 2\mathrm{A}(r) - \mathrm{SA}(X) = 2\mathrm{A}(r) - P(P(X))$
$= \bigcup \mathrm{ASPACE}(\exp_1(dn^r)) - X$, and

(3) $1\mathrm{A} - \mathrm{NESA}(X) = 1\mathrm{A} - \mathrm{SA}(X) = 1\mathrm{A} - P(P(X))$
$= \bigcup \mathrm{ASPACE}(2^{dn}) - X$.

Proof. (1) All inclusions "$\subseteq$" follow from $\mathrm{NESA}(X) \leqslant \mathrm{SA}(X) \leqslant P(P(X))$ (Lemma 3.1) and from Theorem 2.2. Thus it remains to show that, for every $d > 0$, $\mathrm{ASPACE}(\exp_2(ds(n))) - X \subseteq \mathrm{ASPACE}(s(n)) - \mathrm{NESA}(X)$. For $X = X_0$ this is proven in Lemma 5.2 of Ladner, Lipton, and Stockmeyer (1984), and it is obvious that the proof generalizes to arbitrary X (another "free" result). In fact, the proof is very similar to part (i) of the proof of Theorem 2.2; note that the pushdown automaton in that proof uses pop instructions only when comparing successor ID's in its universal

side branches. This comparison is done here in reading mode. The extra exponential can be handled "by preceding each symbol of an ID by a binary address" (Ladner, Lipton, and Stockmeyer, 1984) of length $2^{ds(n)}$, and comparing addresses using $s(n)$-worktape as a counter. For details see Ladner, Lipton, and Stockmeyer (1984).

(2) As in (1). The addresses are of length n^r, and the r heads can be used to count to n^r.

(3) As in (2), observing that, in the proof of Lemma 5.2 of Ladner, Lipton, and Stockmeyer (1984), all comparisons of addresses are done in universal side branches. Note that, upon initialization, the symbols of the first worktape configuration can first be guessed, and then checked against the input in a universal branch. ▮

These results imply the characterizations of iterated stack automata by time complexity classes.

THEOREM 3.5. *For any* $k \geqslant 1$, $s(n) \geqslant \log n$, *and* $r \geqslant 1$,

(1) $\mathrm{NSPACE}(s(n)) - \mathrm{SA}^k = \bigcup \mathrm{DTIME}(\exp_{2k}(ds(n)))$,

(2) $\mathrm{ASPACE}(s(n)) - \mathrm{SA}^k = \mathrm{ASPACE}(s(n)) - \mathrm{NESA}^k = \bigcup \mathrm{DTIME}(\exp_{2k+1}(ds(n)))$,

(3) $2\mathrm{N}(\mathrm{multi}) - \mathrm{SA}^k = \mathrm{DTIME}(\exp_{2k-1}(\mathrm{poly}))$,

(4) $2\mathrm{A}(\mathrm{multi}) - \mathrm{SA}^k = 2\mathrm{A}(\mathrm{multi}) - \mathrm{NESA}^k = \mathrm{DTIME}(\exp_{2k}(\mathrm{poly}))$,

(5) $2\mathrm{N}(r) - \mathrm{SA}^k = \bigcup \mathrm{DTIME}(\exp_{2k-1}(dn^{2r}))$,

(6) $2\mathrm{A}(r) - \mathrm{SA}^k = 2\mathrm{A}(r) - \mathrm{NESA}^k = \bigcup \mathrm{DTIME}(\exp_{2k}(dn^r))$, *and*

(7) $1\mathrm{A} - \mathrm{SA}^k = 1\mathrm{A} - \mathrm{NESA}^k = \bigcup \mathrm{DTIME}(\exp_{2k}(dn))$.

Proof. (2) follows from repeated application of Theorem 3.4(1), ending with an application of Theorem 2.4 and Proposition 2.1 that show that $\mathrm{ASPACE}(s'(n)) = \bigcup \mathrm{DTIME}(\exp_1(ds'(n)))$. Using this, all other equalities follow directly from Theorem 3.2 and Theorem 3.4(2, 3). ▮

The advantage of our "iterated pushdown approach" is that to prove these results we did not have to find any efficient simulation of stack automata (such as in Lemma 5.3 of Ladner, Lipton, and Stockmeyer, 1984). Instead, we simulated stack automata by P^2 automata in a straightforward fashion (Lemma 3.1), and then we just made use twice of the efficient simulation of pushdown automata.

Finally we observe that automata with a storage that is obtained by a mixed application of the operations $P(X)$ and $\mathrm{SA}(X)$ can also be characterized by time complexity classes, using the general theorems on X automata. Thus, e.g., $\mathrm{NSPACE}(s(n)) - \mathrm{SA}(P) = \mathrm{NSPACE}(s(n)) - P(\mathrm{SA}) =$

$\mathrm{ASPACE}(s(n)) - \mathrm{SA} = \mathrm{NSPACE}(s(n)) - P^3 = \bigcup \mathrm{DTIME}(\exp_3(ds(n)))$. The "rule" is: each P gives an exponential jump, alternation (A) gives an exponential jump, and each SA gives a double exponential jump.

4. Iterated Nested Stack Automata

(This section can be skipped by the reader who is not interested in nested stacks.)

A nested stack automaton (Aho, 1969) is a stack automaton that can create (and destroy again) new stacks that are nested inbetween two squares of the old stack. In this way stacks can get nested to any depth. The nested stack can be formalized as a storage type NSA, and, adding an X-configuration to each stack square as usual, as a storage type operation $\mathrm{NSA}(X)$. In this section we show that the nested stack is equivalent to the pushdown of pushdowns. This means that any kind of NSA automaton is equivalent to the corresponding P^2 automaton. In particular it shows that all results on stack automata in the previous section also hold for nested stack automata (thus re-proving and extending some results of Beeri, 1975).

Although we assume the reader to be more or less familiar with nested stack automata, we will give a rather precise, but not too formal, description of the storage type *nested stack of* X, denoted $\mathrm{NSA}(X)$; see Engelfriet and Vogler (1986) for a more formal definition. Let $X = (C, T, F, m, C_0, \mathrm{id})$.

An $\mathrm{NSA}(X)$-configuration consists of stack squares of the form (γ, μ, c) with $\gamma \in \Gamma$, $c \in C$, and $\mu \subseteq \{\not\!\!\$, \$, \uparrow\}$, where $\not\!\!\$$, $\$$, and $\uparrow$ are symbols not in Γ. Intuitively, $\uparrow \in \mu$ means that the stack pointer points to the square, $\not\!\!\$ \in \mu$ means that it is a bottom square (of one of the nested stacks), and $\$ \in \mu$ means that it is a top square. To be more precise, an $\mathrm{NSA}(X)$-configuration c' is a sequence

$$c' = (\gamma_0, \mu_0, c_0)(\gamma_1, \mu_1, c_1) \cdots (\gamma_n, \mu_n, c_n)$$

of such stack squares (with $n \geqslant 0$), satisfying the following requirements:

(i) $\not\!\!\$ \in \mu_0$ and $\$ \in \mu_n$,

(ii) there is exactly one j, $0 \leqslant j \leqslant n$, such that $\uparrow \in \mu_j$; this unique integer will be indicated by i in what follows (thus the stack pointer points to square (γ_i, μ_i, c_i)), and

(iii) $\$ \notin \mu_j$ for all $0 \leqslant j < i$.

Another requirement on c' that will automatically be satisfied for any

NSA(X)-configuration that is actually used by an automaton is that all occurrences of ¢ and \$ in c' are well nested, viewing ¢ as a left parenthesis and \$ as a right parenthesis (and if μ_j contains both ¢ and \$, then ¢ is supposed to occur before \$). This requirement in fact expresses the nested stack structure: every pair of matching parentheses ¢ and \$ corresponds to a stack.

In Fig. 3(a) a picture is shown of a nested stack configuration $(\gamma_1, \{¢\}, c_1)(\gamma_{10}, \{¢, \uparrow\}, c_{10})(\gamma_{11}, \{\$\}, c_{11})(\gamma_2, \varnothing, c_2)(\gamma_9, \{¢, \$\}, c_9)(\gamma_5, \{¢\}, c_5)(\gamma_6, \varnothing, c_6)(\gamma_8, \{¢, \$\}, c_8)(\gamma_7, \{\$\}, c_7)(\gamma_3, \varnothing, c_3)(\gamma_4, \{\$\}, c_4)$. The reason for this numbering will appear below. In each square (γ_j, μ, c_j) of Fig. 3(a), μ and j are shown. The lines underneath indicate the nesting structure of the five stacks involved; in parentheses this structure is (( )( )(( ))). Taken apart, these five stacks are

$$s_1 = (\gamma_1, \{¢\}, c_1)(\gamma_2, \varnothing, c_2)(\gamma_3, \varnothing, c_3)(\gamma_4, \{\$\}, c_4),$$
$$s_2 = (\gamma_5, \{¢\}, c_5)(\gamma_6, \varnothing, c_6)(\gamma_7, \{\$\}, c_7),$$
$$s_3 = (\gamma_8, \{¢, \$\}, c_8),$$
$$s_4 = (\gamma_9, \{¢, \$\}, c_9), \text{ and}$$
$$s_5 = (\gamma_{10}, \{¢, \uparrow\}, c_{10})(\gamma_{11}, \{\$\}, c_{11}).$$

The initial NSA(X)-configurations are of the form $(\gamma_0, \{¢, \$, \uparrow\}, c_0)$ with $\gamma_0 \in \Gamma$ and $c_0 \in C_0$. Apart from the identity, the instructions of NSA(X) are push(γ, f), pop, move-down, move-up, create(γ), and destroy. We now discuss the effect of their execution on the above NSA(X)-configuration c'. Let c'' denote the resulting NSA(X)-configuration (if it is defined). Recall that i is the number of the square pointed at by the stack pointer.

— push(γ, f) is defined on c' only if $\$ \in \mu_i$ and $m(f)(c_i)$ is defined. To obtain c'' from c', replace (γ_i, μ_i, c_i) by $(\gamma_i, \mu_i - \{\$, \uparrow\}, c_i)(\gamma, \{\$, \uparrow\}, m(f)(c_i))$.

— pop is defined on c' only if $\$ \in \mu_i$ and $¢ \notin \mu_i$ (there are no empty stacks). To obtain c'' from c', replace $(\gamma_{i-1}, \mu_{i-1}, c_{i-1})(\gamma_i, \mu_i, c_i)$ by $(\gamma_{i-1}, \mu_{i-1} \cup \{\$, \uparrow\}, c_{i-1})$.

— move-down is defined only if $i > 0$. Replace $(\gamma_{i-1}, \mu_{i-1}, c_{i-1})(\gamma_i, \mu_i, c_i)$ by $(\gamma_{i-1}, \mu_{i-1} \cup \{\uparrow\}, c_{i-1})(\gamma_i, \mu_i - \{\uparrow\}, c_i)$.

— move-up is defined only if $\$ \notin \mu_i$. Replace $(\gamma_i, \mu_i, c_i)(\gamma_{i+1}, \mu_{i+1}, c_{i+1})$ by $(\gamma_i, \mu_i - \{\uparrow\}, c_i)(\gamma_{i+1}, \mu_{i+1} \cup \{\uparrow\}, c_{i+1})$.

— create(γ) is always defined. Replace (γ_i, μ_i, c_i) by $(\gamma, \{¢, \$, \uparrow\}, c_i)(\gamma_i, \mu_i, c_i)$. Thus, a new one-square stack is nested below the square pointed at, with stack symbol γ, and with the same X-configuration. The stack pointer moves to the new stack.

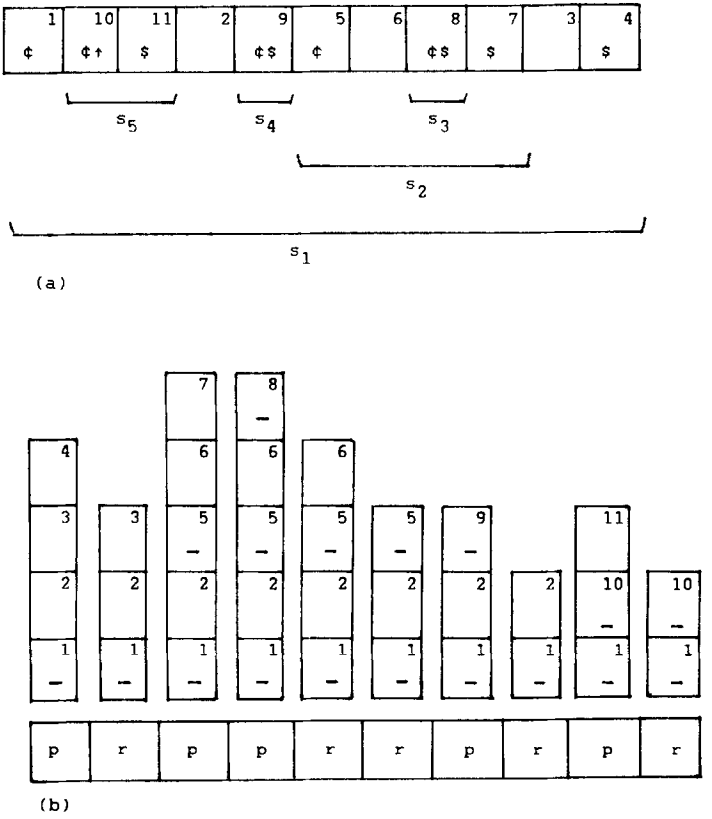

FIG. 3. Simulation of a nested stack by a pushdown of pushdowns.

— destroy is defined only if $\mu_i = \{¢, \$, \uparrow\}$ and $n > 0$. Replace $(\gamma_i, \mu_i, c_i)(\gamma_{i+1}, \mu_{i+1}, c_{i+1})$ by $(\gamma_{i+1}, \mu_{i+1} \cup \{\uparrow\}, c_{i+1})$. Thus, the nested stack is destroyed, and the stack pointer returns to the square the stack was created from.

The tests of NSA(X) are sym $= \gamma$ and test(t), where sym $= \gamma$ tests whether $\gamma_i = \gamma$, and test(t) tests whether $m(t)(c_i) =$ true.

In the example of Fig. 3(a), s_2 was created from square 3 of s_1, s_3 from square 7 of s_2, s_4 from square 5 of s_2, and s_5 from square 2 of s_1. Any nested stack automaton necessarily has to create these stacks in this order, due to the fact that it cannot move-up past a \$ (thus, the order of creation of stacks is the order, from right to left, of their top squares). This means that the squares are numbered in the order in which they have come into life during the computation of an NSA(X) automaton.

This ends the description of NSA(X). Since SA(X) is obtained by

dropping the create(γ) and destroy instructions from NSA(X), SA(X) $\leqslant$ NSA(X). The definition of NSA$^k(X)$ and NSA^k is as for P and SA.

The equivalence of NSA(X) and $P(P(X))$, extending Lemma 3.1, was already proven in Section 7 of Engelfriet and Vogler (1986)(and, as noted in the introduction, the original idea came from Aho and Ullman). Here we give a proof that is easier to read, due to our simpler definition of equivalence (in Section 1.2). Note that we have to reprove this equivalence anyway, because the "Justification Theorem" (Theorem 1.2.4) was not proven for the automaton types of this paper in Engelfriet and Vogler (1986).

THEOREM 4.1. *For every storage type* X, NSA(X) $\equiv P(P(X))$.

Proof. We have to show that NSA(X) $\leqslant P(P(X))$ and $P(P(X)) \leqslant$ NSA(X).

(1) Let us start with the simulation of a nested stack storage by a pushdown of pushdowns, extending the proof of Lemma 3.1. Let M be a 1-way deterministic NSA(X) transducer, with stack alphabet $\Gamma_M \subseteq \Gamma$. A 1-way deterministic $P_s(P_s(X))$ transducer M' equivalent to M can be constructed as follows (note that we allow M' to use stay(γ, f) instructions on both levels). As in Engelfriet and Vogler (1986), the simulation of a nested stack by a pushdown of pushdowns is based on the obvious fact that every NSA(X)-configuration used by M can be built up from the initial configuration $(\gamma_0, \{\not\in, \$, \uparrow\}, c_0)$ of M by a sequence of push, move-down, and create instructions. This sequence is unique (apart from possible nonuniqueness caused by the instructions f in push(γ, f) instructions). The $P(P(X))$ – configuration used by M' to simulate this NSA(X)-configuration is obtained by executing a corresponding sequence of $P(P(X))$ instructions to the initial configuration (p, $(\bar{\gamma}_0, c_0)$) of M', where push(γ, f) corresponds to stay(p, push(γ, f)), move-down to push(r, pop), just as for SA(X) in Lemma 3.1, and create(γ) to push(p, stay($\bar{\gamma}$, id)). As in Lemma 3.1, we use p and r on the "outer" pushdown to indicate the pushdown mode ($\$ \in \mu_i$) and the reading mode ($\$ \notin \mu_i$) of M, respectively. Moreover, in the inner pushdowns we use barred symbols $\bar{\gamma}$ to indicate bottom squares of the NSA(X)-configuration (for every $\gamma \in \Gamma_M$, $\bar{\gamma}$ is a new symbol in Γ). As an example, Fig. 3(b) pictures the $P(P(X))$-configuration corresponding to the NSA(X)-configuration of Fig. 3(a). It is shown in each square of the $P(P(X))$-configuration whether or not the stack symbol is barred. The number of a square indicates that the (unbarred) stack symbol and the X-configuration are the same as those in the corresponding square of the NSA(X)-configuration. Note also that the concatenation of all top squares of all inner pushdowns equals the reverse

of all squares of the NSA(X)-configuration to the right of (and including) the square pointed at.

This description should suffice to understand that M' can simulate the instructions and tests of M as follows. Each NSA(X) instruction or test to the left is simulated by $P(P(X))$ instructions and tests as indicated on the right.

push(γ, f)	stay(p, push(γ, f))	provided top = p
pop	stay(p, pop)	provided top = p and test(top = γ) for some (unbarred) $\gamma \in \Gamma_M$
move-down	push(r, pop)	
move-up	pop	provided top = r
create(γ)	push(p, stay($\bar{\gamma}$, id))	
destroy	pop	provided top = p and test(top = $\bar{\gamma}$) for some $\gamma \in \Gamma_M$
sym = γ	test(top = γ) or test(top = $\bar{\gamma}$)	
test(t)	test(test(t))	

As noted before, if $(\gamma_0, \{\not\in, \$, \uparrow\}, c_0)$ is the initial configuration of M, then $(p, (\bar{\gamma}_0, c_0))$ is the one of M'. The state behaviour and the input and output of M are simulated by M' in the obvious way.

(2) Next we show how to simulate a pushdown of pushdowns by a nested stack. Let M be a 1-way deterministic $P(P(X))$ transducer. To start with we observe that M can be transformed in such a way that the symbols on its "outer" pushdown are all the same. In fact it is easy to code each symbol of the outer pushdown into the symbol part of the top square of the corresponding inner pushdown (note that empty pushdowns do not exist). The top = γ test can be simulated by appropriate test(top = γ') tests.

Thus we may assume that M just uses one symbol on its outer pushdown. We will denote this symbol by "–". The simulation of M by a 1-way deterministic NSA(X) transducer M' is based on the fact that every $P(P(X))$-configuration c used by M can be built up from the initial configuration $(-, (\gamma_0, c_0))$ by a sequence of push(–, push(γ, f)), push(–, pop), and push(–, id) instructions. As in part (1) of this proof, this sequence is unique (apart from the f's). M' simulates this $P(P(X))$-configuration by the NSA(X)-configuration that is obtained from its initial configuration (γ_0, c_0) by executing the corresponding sequence of NSA(X) instructions. In this sequence push(–, push(γ, f)) corresponds to (create(β); push(γ, f)), push(–, pop) to move-down, and push(–, id) to create(β), where β is the

symbol such that test(top $= \beta$) holds before execution of the $P(P(X))$ instruction. In this way the top-most inner pushdown of the $P(P(X))$-configuration contains precisely all squares of the NSA(X)-configuration that are to the left of (and including) the square pointed at. As an example, the $P(P(X))$-configuration of Fig. 4(a) is built up from the initial configuration by a sequence of instructions of the form push(–, push(γ_1, f_1)); push(–, push(γ_2, f_2)); push(–, id); push(–, pop). The corresponding NSA(X)-configuration is shown in Fig. 4(b). Two squares in Fig. 4 are given the same number to represent that they contain the same stack symbol and the same X-configuration. Note that the stacks s_1, s_2, s_3, and s_4 correspond to the first four inner pushdowns (i.e., those that are not obtained by a push(–, pop) instruction). Note also that all stacks are of size 1 or 2 (without the inner stacks).

Of course, M' simulates the above-mentioned push instructions of M as indicated above. The only remaining $P(P(X))$ instruction is pop. This can be simulated by M' because it can reverse the effect of a simulated push(–, ...) instruction by distinguishing between the above three possibilities. To do this, it needs additional tests "bottom" and "top" that are true iff the square pointed at is a bottom square or a top square, respectively. It should be clear that we may allow these tests (formally, it can be shown that the resulting storage type is equivalent to NSA(X)). The pop instruction of M is now simulated by M' as follows:

if not top **then** move-up
else if not bottom **then** (pop; destroy)
else destroy.

The three lines in this program correspond to the case that the last $P(P(X))$ instruction in the sequence discussed above is a push(–, pop), push(–, push(γ, f)), or push(–, id) instruction, respectively.

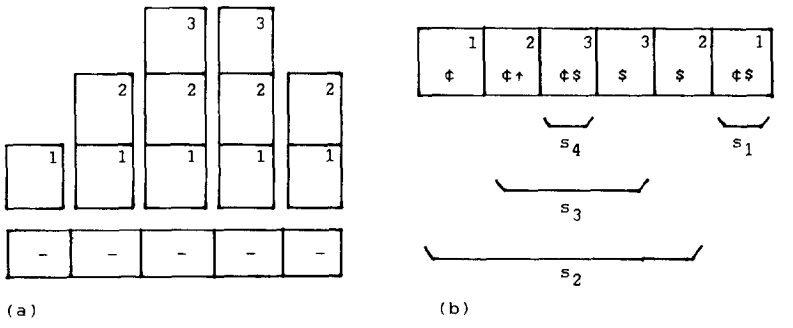

FIG. 4. Simulation of a pushdown of pushdowns by a nested stack.

Finally, a test test(top $=\gamma$) is simulated by the test sym $=\gamma$, and a test test(test(t)) by the test test(t). Note that the test top $=-$ needs no simulation.

This should convince the reader that M' can simulate M. ▮

It would not be difficult to prove the monotonicity of the operation NSA(X) with respect to $\leqslant$, along the lines of the proof of Theorem 1.3.1. However, monotonicity of NSA(X) follows directly from Theorem 4.1 and the monotonicity of $P(X)$ (shown in Theorem 1.3.1). Theorem 4.1 also implies the following two corollaries.

COROLLARY 4.2. *For every* $k \geqslant 0$, $\mathrm{NSA}^k \equiv P^{2k}$.

Proof. Assume $\mathrm{NSA}^k \equiv P^{2k}$. Then, since $P(X)$ preserves equivalence (Theorem 1.3.1), $P(P(\mathrm{NSA}^k)) \equiv P^{2k+2}$. By Theorem 4.1, for $X = \mathrm{NSA}^k$, $P(P(\mathrm{NSA}^k)) \equiv \mathrm{NSA}^{k+1}$. ▮

COROLLARY 4.3. *For every automaton type* Y *and every* $k \geqslant 1$, $Y - \mathrm{NSA}^k = Y - P^{2k}$.

Proof. By Corollary 4.2 and Theorem 1.2.4. ▮

Note that this corollary includes one-way nondeterministic automata and deterministic automata (and transducers).

It can be concluded that in all results of Section 3, SA may be replaced by NSA. Thus, for those types of automata, stacks, nested stacks, and pushdowns of pushdowns all have the same power.

We observed before that SA cannot always be replaced by P^2 because it is not true that $P^2 \leqslant \mathrm{SA}$. Indeed, it is not true that $\mathrm{NSA} \leqslant \mathrm{SA}$, because $1\mathrm{N} - \mathrm{SA} \subsetneq 1\mathrm{N} - \mathrm{NSA}$ (see p. 27 of Greibach, 1970).

5. ITERATED CHECKING STACK AUTOMATA

A checking stack automaton (Greibach, 1969) is a nonerasing stack automaton that is not allowed to push after it has executed a move-down instruction. Thus, it is not allowed to change from reading mode to pushdown mode. It is easy to define this formally as a storage type CSA, by adding an extra component to the configurations of NESA that indicates whether the configuration is in pushdown mode or in reading mode. We leave the details to the reader. Similarly, the operation CSA(X) can be defined, and it can easily be shown that $\mathrm{CSA}(X) \leqslant \mathrm{NESA}(X)$. As usual, CSA($X$) is monotonic, and $\mathrm{CSA}^k(X)$ and CSA^k are defined as usual.

In this section we show that nondeterministic multi-head iterated checking stack automata characterize iterated exponential *space* complexity

classes (as one would expect from Fischer, 1969, and Ibarra, 1971). The results of this section will be applied to the one-way languages in Section 7. Moreover, they give more information on NESA automata.

The characterization will follow from the next result, by induction.

THEOREM 5.1. *For any storage type X and $s(n) \geqslant \log n$,* $\mathrm{NSPACE}(s(n)) - \mathrm{CSA}(X) = \mathrm{NSPACE}(s(n)) - \mathrm{NESA}(X) = \bigcup \mathrm{NSPACE}(2^{ds(n)}) - X$.

Proof. Let $X = (C, T, F, m, C_0, \mathrm{id})$.

(i) We first show that for any $d > 0$ $\mathrm{NSPACE}(2^{ds(n)}) - X \subseteq \mathrm{NSPACE}(s(n)) - \mathrm{CSA}(X)$. This is again a variation of the proof of Theorem 2.2, or more precisely of the proof of Theorem 3.2(1), where it is shown that $\mathrm{ASPACE}(2^{ds(n)}) - X \subseteq \mathrm{NSPACE}(s(n)) - \mathrm{SA}(X)$. Note that in our case, since there is no alternation, there is no need to backtrack (through the computation tree of M). Hence the stack is nonerasing, which already shows that $\mathrm{NSPACE}(2^{ds(n)}) - X \subseteq \mathrm{NSPACE}(s(n)) - \mathrm{NESA}(X)$. However, M' may as well start by guessing nondeterministically the whole ultimate contents of its stack (after it has copied the input to the stack, initially), and then check that (α_{k+1}, c_{k+1}) is the correct successor of (α_k, c_k) for all k, from left to right. Note that M' can count to $2^{ds(n)}$ using its worktape, and thus can walk back and forth between two consecutive configurations of M. In order to check that the X-configuration part of the checking stack corresponds to the moves of M, M' should record the instructions $f \in F$ which it has applied during the guessing phase. Thus, M' should use instructions $\mathrm{push}(\langle m_i, f\rangle, f)$ rather than $\mathrm{push}(m_i, f)$, where $\langle m_i, f\rangle$ is a stack symbol. Tests on X-configurations can of course still be executed by M' during the checking phase. This shows that M' can be constructed as an $\mathrm{NSPACE}(s(n)) - \mathrm{CSA}(X)$ automaton.

(ii) Second, we have to show that $\mathrm{NSPACE}(s(n)) - \mathrm{NESA}(X) \subseteq \bigcup \mathrm{NSPACE}(2^{ds(n)}) - X$. "Unfortunately," we are now forced to simulate a stack automaton, without help from our results on pushdown automata (cf. the end of Section 3). Fortunately, however, we only need the standard technique of transition tables (Section 14.2 of Hopcroft and Ullman, 1979). Let M be an $\mathrm{NSPACE}(s(n)) - \mathrm{NESA}(X)$ automaton. We may assume that M accepts when the stack pointer is at the top of the stack. We have to construct an $\mathrm{NSPACE}(2^{ds(n)}) - X$ automaton M' equivalent to M. M' simulates M by keeping track of the top square of the stack of M, and keeping track of a transition table to represent the rest of that stack. The symbol part of the top square can of course be kept in the finite control of M', whereas the X-configuration part is kept as the X-configuration of M'. The transition table is kept on the worktape, together with the worktape contents of M.

We now explain the notion of transition table. For fixed input string, let

us call the part α of an ID (α, c') of M, where c' is a NESA(X)-configuration, the "worktape configuration" of M; α includes the worktape contents and position of the worktape head, and also the state and the position of the input head, but not the input tape. Thus, α can be coded as a string of size $O(s(n))$. Now, if c' is a NESA(X)-configuration with the stack pointer at the top square, then the corresponding transition table $R(c')$ is the set of pairs of worktape configurations, defined as follows: $\langle \alpha, \beta \rangle \in R(c')$ if and only if there is a computation $(\alpha, c'_1) \vdash^* (\beta, c')$ of M such that c' occurs in the last ID of the computation only, where c'_1 is obtained from c' by executing one move-down instruction. Thus the transition table completely characterizes all the computations that M can do in reading mode. If M' knows $R(c')$, it additionally only needs to keep the top square of c' to be able to simulate the behaviour of M. As soon as M executes a move-down instruction, M' consults $R(c')$ and shortcuts the reading mode computation of M. Clearly, the size of the transition table is $O(2^{ds(n)})$ for some $d > 0$, and thus fits into the workspace of M'. M' can keep track of the current worktape configuration of M by marking it in the transition table. It remains to discuss how M' simulates a push(γ, f) instruction of M. First it updates its transition table. The new transition table can be computed from the old transition table and the old top square, by checking nondeterministically, for every pair of worktape configurations, whether there exists a computation from one to the other as described above. As soon as the computation becomes longer than the number of worktape configurations (where each application of the old transition table counts as a step), M' can stop the check because a repetition occurred. It should be clear that M' can do all this in space $O(2^{ds(n)})$. Second, M' updates the top square by storing γ in its finite control, and executing f on its X-storage. This ends the description of M'. We repeat that the above is just a standard technique that turns out to work fine also for X automata. ▮

For r-head automata the proofs are more subtle. It may not be sufficient to use transition tables (for CSA), and if it is (for NESA), it is more difficult to see that they can be updated within the proper space (see Hopcroft and Ullman, 1967b).

From this theorem the characterization of the iterated exponential space complexity classes by iterated checking stack automata follows.

THEOREM 5.2. *For any* $k \geq 1$, $\mathrm{2N(multi)} - \mathrm{CSA}^k = \mathrm{2N(multi)} - \mathrm{NESA}^k = \mathrm{DSPACE}(\exp_{k-1}(\mathrm{poly}))$.

Proof. Iterated appliation of Theorem 5.1 gives that $\mathrm{NSPACE}(\log n) - \mathrm{CSA}^k = \mathrm{NSPACE}(\log n) - \mathrm{NESA}^k = \mathrm{NSPACE}(\exp_{k-1}(\mathrm{poly}))$. The result now follows from the fact that $\mathrm{NSPACE}(\log n) - X = \mathrm{2N(multi)} - X$

for every storage type X, and from Savitch's theorem ($\mathrm{NSPACE}(s(n)) \subseteq \mathrm{DSPACE}(s(n)^2)$; see Hopcroft and Ullman, 1979). ▮

A comparison with Theorem 3.5(4) shows that alternation gives a big jump in power to multi-head NESA^k automata.

Note that, by Theorem 5.1, Proposition 2.1, and Theorem 2.5, $\mathrm{NSPACE}(s(n)) - \mathrm{CSA}^k(P) = \mathrm{NSPACE}(s(n)) - P^{k+1}$. One would not expect this when considering the storage types $\mathrm{CSA}^k(P)$ and P^{k+1} on their own (CSA and P are incomparable storage types, with respect to $\leqslant$).

6. Iterated Stack-Pushdown Automata

(This section can be skipped by the reader, who should then also skip the part of Section 7.1 following Theorem 7.8.)

Checking stack-pushdown automata were introduced in van Leeuwen (1976), where the corresponding auxiliary automata were used to obtain a uniform characterization of certain well-known complexity classes. In Engelfriet, Schmidt, and van Leeuwen (1980) they were generalized in the obvious way to stack-pushdown automata (and compared to macro grammars; see also Engelfriet and Slutzki, 1984).

A stack-pushdown (SPD) automaton is a stack automaton with an additional pushdown. The stack and the pushdown are not independent, but should satisfy the restriction that the length of the pushdown is equal to the number of squares above the stack pointer (including the square pointed at). Thus, if one imagines the pushdown upside down with its bottom next to the top of the stack, the top of the pushdown has to follow the movements of the stack pointer, cf. Fig. 5. This restriction is formalized by giving the storage type SPD the same instructions as SA except that move-down is replaced by move-down(γ), where γ is a pushdown symbol that should be pushed on the pushdown. Execution of the move-up instruction automatically involves a pop of the pushdown. SPD has tests $\mathrm{sym} = \gamma$ and $\mathrm{top} = \gamma$, where sym refers to the stack symbol and top to the pushdown symbol (both taken from Γ). Thus, the pushdown can only be used in reading mode. In pushdown mode it always contains one square. An initial configuration of SPD consists of a one-square stack and a one-square pushdown.

We now define the storage type operation *stack-pushdown of* X, denoted $\mathrm{SPD}(X)$, by adding X-configurations to the stack squares only; the pushdown just has symbols in its squares. (This is just the opposite of, and should not be confused with, the storage type operation of the same name studied in Engelfriet and Slutzki, 1984). Thus $\mathrm{SPD}(X)$ has instructions push(γ, f), pop, move-down(γ), move-up, and id, and it has tests $\mathrm{sym} = \gamma$,

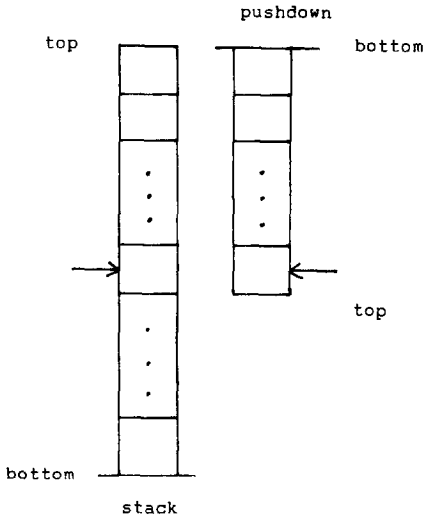

FIG. 5. A stack-pushdown configuration.

top $= \gamma$, and test(t). The details are left to the reader. We stress again that push and pop operate on the stack (when in pushdown mode), whereas move-down(γ) and move-up push and pop the pushdown, respectively. By appropriately restricting the stack of SPD(X), the storage type CSPD(X) is obtained: the checking stack-pushdown of X. Note that CSA(X) $\leqslant$ CSPD(X) and SA(X) $\leqslant$ SPD(X).

LEMMA 6.1. *For every storage type* X, SPD(X) $\leqslant P(P(X))$.

Proof. The simulation is exactly the same as in Lemma 3.1. The symbols of the pushdown can be stored in the symbol part of the "outer" pushdown, in addition to the p and r markers. ∎

Since SA(X) $\leqslant$ SPD(X) $\leqslant P(P(X))$, it follows from Section 3 that all results stated there for SA also hold for SPD. This takes care of iterated SPD automata.

Next we show that the analogues of Theorems 5.1 and 5.2 hold for CSPD automata.

THEOREM 6.2. *For any storage type* X *and* $s(n) \geqslant \log n$, NSPACE($s(n)$) $-$ CSPD(X) $= \bigcup$ NSPACE($2^{ds(n)}$) $- X$.

Proof. Since CSA(X) $\leqslant$ CSPD(X), it suffices to generalize part (ii) of the proof of Theorem 5.1. Let M be an NSPACE($s(n)$) $-$ CSPD(X) automaton. An NSPACE($2^{ds(n)}$) $- X$ automaton M' that simulates M can

be constructed in exactly the same way. M' keeps track of the top square of the stack, now including the bottom square of the pushdown (in its finite control). The rest of the stack can be represented by a finite set of transition tables, one for each $\gamma \in \Gamma$ (used in the first move-down(γ) instruction). Thus, taking over the notation in the proof of Theorem 5.1, if c' is a CSPD(X)-configuration with the stack pointer at the top square, then, for each $\gamma \in \Gamma$, the corresponding transition table $R_\gamma(c')$ is defined by $\langle \alpha, \beta \rangle \in R_\gamma(c')$ iff there is a computation $(\alpha, c'_\gamma) \vdash^* (\beta, c')$ of M such that c' occurs in the last ID of the computation only, where c'_γ is obtained from c' by executing move-down(γ). The rest of the proof is analogous. ▮

As a corollary we obtain the analogue of Theorem 5.2.

THEOREM 6.3. *For any* $k \geq 1$, $2\text{N(multi)} - \text{CSPD}^k = \text{DSPACE}(\exp_{k-1}(\text{poly}))$.

7. APPLICATIONS TO ONE-WAY AUTOMATA

In this final section we apply our results to formal language theory. Characterizations of multi-head automata (as obtained in the previous sections) can be used to prove proper inclusions between classes of languages accepted by one-way automata, and to provide (upper and lower) bounds on the complexity of the emptiness problem for one-way automata. These two applications will be discussed one by one.

7.1. *Hierarchies of One-Way Iterated Automata*

The first application consists in particular of showing that the one-way iterated pushdown automata form a proper hierarchy at each level, i.e., that one-way $(k+1)$-iterated pushdown automata are more powerful than one-way k-iterated pushdown automata for every $k \geq 1$ (and similarly for iterated stack, checking stack, etc., automata). The time/space complexity characterization of the 2-way multi-head iterated X automata (where X = pushdown, stack, checking stack, etc.) implies (by the usual time/space hierarchy theorems) that these automata form a proper hierarchy at each level of iteration. The following straightforward result allows us to "translate" these hierarchies "down" to 1-way iterated X automata. The "translation" is performed by 2N(multi) transducers, i.e., by non-deterministic log-space reductions. For a class of transductions YT and a class of languages K we denote by $YT^{-1}(K)$ the class of languages $\{\tau^{-1}(L) \mid \tau \in YT,\ L \in K\}$.

LEMMA 7.1. *For every storage type* X, $2N(multi) - X = 2N(multi)\, T^{-1}(1N - X) = 2N(multi)\, T^{-1}(1D - X)$.

Proof. The proof uses standard techniques from automata theory. Let $X = (C, T, F, m, C_0, id)$.

(i) To show that $2N(multi) - X \subseteq 2N(multi)\, T^{-1}(1D - X)$, let M be a 2-way nondeterministic multi-head X automaton, that uses tests $T_M \subseteq T$ and instructions $F_M \subseteq F$, and has the initial X-configuration c_0. The idea of the decomposition of M (familiar from AFA theory, Ginsburg, 1975) is to turn M into a transducer M' that does not execute the tests and instructions of M on storage, but rather prints them on its output tape. A 1-way deterministic X automaton N is then used to check the "executability" of that sequence of tests and instructions. This is a variation of the proof of Theorem 1.2.4.

The 2-way nondeterministic multi-head finite transducer M' has the same states, initial state, final states, input alphabet, and number of heads as M. The output alphabet of M' is $\Omega = R(T_M) \cup F_M$. If the transition function δ of M has the form $\delta\colon A \times R(T_M) \to P_{fin}(B \times F_M)$, cf. Section 1.1, then the transition function δ' of M' has the form $\delta'\colon A \to P_{fin}(B \times \Omega^*)$. In fact, for $a \in A$, $\delta'(a) = \{(b, \rho f) \mid \delta(a, \rho) \text{ contains } (b, f)\}$, where ρf is a string of length 2 over the alphabet Ω. Thus, M' simulates M by guessing the test results of its X-configurations. These guesses will be checked by feeding the output of M' into the automaton N. N has input alphabet Ω and initial X-configuration c_0, and it accepts a string $\rho_1 f_1 \rho_2 f_2 \cdots \rho_n f_n$ if and only if $m(f_1 \cdots f_n)(c_0)$ is defined and $m(\rho_1)(c_0) = \text{true}$ and $m(\rho_{i+1})(m(f_1 \cdots f_i)(c_0)) = \text{true}$ for all $1 \leqslant i \leqslant n-1$. The transition function of N is a partial function $\delta_N\colon Q \times \Omega \times R(T_M) \to Q \times F_M$, where Q consists of one (initial and final) state q. For every $\rho \in R(T_M)$ and $f \in F_M$, $\delta_N(q, \rho, \rho) = (q, id)$ and $\delta_N(q, f, \rho) = (q, f)$. The other values of δ_N are undefined. It should now be clear that $L(M) = \tau(M')^{-1}(L(N))$.

(ii) To show that $2N(multi)\, T^{-1}(1N - X) \subseteq 2N(multi) - X$, let M be a $2N(multi)$ transducer and N a $1N - X$ automaton. A $2N(multi) - X$ automaton M' such that $L(M') = \tau(M)^{-1}(L(N))$ can be obtained from M and N by a straightforward product construction. We may assume that M outputs strings of length at most one at each step. M' simulates all output-less moves of M and all input-less moves of N separately. As soon as M produces an output symbol, M' feeds this symbol into N; thus, in this case, it simulates a move of M and a move of N simultaneously. ∎

This theorem holds in fact for any nondeterministic automaton type Y (without T): $Y - X = YT^{-1}(1N - X) = YT^{-1}(1D - X)$. It might be called the "inverse law" of automata theory, and it enables us to translate proper inclusions between $Y - X$ automata down to proper inclusions between

one-way X automata. In our case we use this law to translate the fact that "$k+1$ iterations are more than k" from multi-head to one-way automata.

COROLLARY 7.2. *Let U be an operation on storage types, such that $U^k \leqslant U^{k+1}$ for every $k \geqslant 1$ (where, as usual, U^k abbreviates $U^k(X_0)$). For every $k \geqslant 1$, if $2N(multi) - U^k \subsetneq 2N(multi) - U^{k+1}$, then $1N - U^k \subsetneq 1N - U^{k+1}$, and even $1D - U^{k+1} \not\subseteq 1N - U^k$.*

Proof. Assume that $1D - U^{k+1} \subseteq 1N - U^k$. Then $2N(multi)\, T^{-1}(1D - U^{k+1}) \subseteq 2N(multi)\, T^{-1}(1N - U^k)$. Hence, by Lemma 7.1, $2N(multi) - U^{k+1} \subseteq 2N(multi) - U^k$. ∎

As we have seen, for all operations U considered, $2N(multi) - U^k$ is a time or space complexity class. Since it is well known that these complexity classes are properly included in each other (for growing k), Corollary 7.2 implies that the $1N - U^k$ automata form a proper hierarchy. Note that the proper inclusion of complexity classes is proven by a diagonalization argument. Thus, in a way, we prove $1N - U^k \subsetneq 1N - U^{k+1}$ also by diagonalization, which is a technique that usually does not work for one-way automata! For completeness's sake we state the needed proper inclusions of complexity classes (which follow from the time and space hierarchy theorems, see Hopcroft and Ullman, 1979).

PROPOSITION 7.3. *For every $k \geqslant 0$, $DTIME(\exp_k(poly)) \subsetneq DTIME(\exp_{k+1}(poly))$, and $DSPACE(\exp_k(poly)) \subsetneq DSPACE(\exp_{k+1}(poly))$.*

We are now able to state the first main result of this section: properness of the hierarchy of one-way iterated pushdown languages.

THEOREM 7.4. *The diagram of Fig. 6 is correct (i.e., ascending lines denote proper inclusions, and classes that are not connected by ascending lines are incomparable). In formulas this means that, for every $k \geqslant 1$, $1N - P^k \subsetneq 1N - P^{k+1}$ and even $1D - P^{k+1} \not\subseteq 1N - P^k$. Moreover, $1N - P \not\subseteq \bigcup_k 1D - P^k$.*

Proof. The inclusions are obvious. The noninclusion $1D - P^{k+1} \not\subseteq 1N - P^k$ follows from Theorem 2.6 (the time complexity characterization of $2N(multi) - P^k$), Proposition 7.3 (which implies that $\{2N(multi) - P^k\}$ is a proper hierarchy at each level), and Corollary 7.2 (that translates this result down to $1N - P^k$). The existence of a context-free language not in $\bigcup_k 1D - P^k$ is shown in Section 4 of Engelfriet and Vogler (1987). ∎

In Damm and Goerdt (1986) (see also Engelfriet and Vogler, 1988) it is shown that $1N - P^k = k - OI$, where $k - OI$ is the class of languages generated by k-level OI macro grammars, i.e., the kth class in the well-

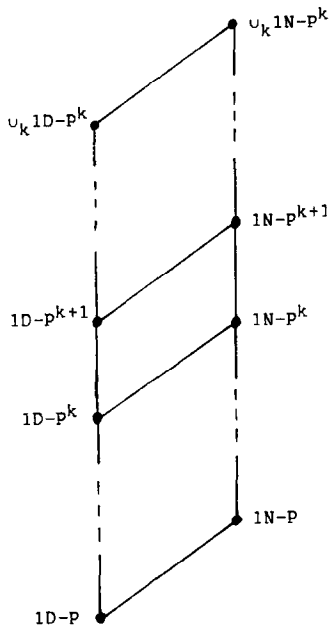

FIG. 6. The hierarchy of one-way iterated pushdown languages.

known OI-hierarchy (Wand, 1975; Maibaum, 1974; Engelfriet and Schmidt, 1977/1978; Damm, 1982; Vogler, 1988). Damm (1982) proved that $k-\mathrm{OI} \subsetneq (2k+1)-\mathrm{OI}$, by the method of rational index. Theorem 7.4 shows that in fact $k-\mathrm{OI} \subsetneq (k+1)-\mathrm{OI}$, i.e., the OI-hierarchy is proper at each level.

Let us now consider some of the other iterated storage types. For the nested stack the situation is clear, because, by Corollary 4.3, $1\mathrm{N}-\mathrm{NSA}^k = 1\mathrm{N}-P^{2k}$ and $1\mathrm{D}-\mathrm{NSA}^k = 1\mathrm{D}-P^{2k}$. For the iterated stack automata it can be shown that $1\mathrm{N}-\mathrm{SA}^k \subsetneq 1\mathrm{N}-\mathrm{SA}^{k+1}$ in exactly the same way as in the proof of Theorem 7.4, using the time complexity characterization of $2\mathrm{N}(\mathrm{multi})-\mathrm{SA}^k$ in Theorem 3.5(3). The relationship of the SA-hierarchy to the P-hierarchy is less clear, but the following can be said. From the fact that, for every X, $P(X) \leqslant \mathrm{SA}(X) \leqslant P(P(X))$, see Lemma 3.1, it easily follows by induction (using the monotonicity of P) that $P^k \leqslant \mathrm{SA}^k \leqslant P^{2k}$ for every $k \geqslant 1$. Hence, by Theorem 1.2.4, $1\mathrm{N}-P^k \subseteq 1\mathrm{N}-\mathrm{SA}^k \subseteq 1\mathrm{N}-P^{2k}$. This shows, of course, that the 1-way iterated stack languages are the same as the 1-way iterated pushdown languages, i.e., $\bigcup_k 1\mathrm{N}-\mathrm{SA}^k = \bigcup_k 1\mathrm{N}-P^k$. Our translation technique can be used to show that the inclusion $1\mathrm{N}-\mathrm{SA}^k \subseteq 1\mathrm{N}-P^{2k}$ is optimal, i.e., $1\text{N-SA}^k \not\subseteq 1\mathrm{N}-P^{2k-1}$. In fact, even the inclusion $1\mathrm{D}-\mathrm{SA}^k \subseteq 1\mathrm{N}-P^{2k-1}$ would imply by Lemma 7.1 that

$2N(multi)-SA^k \subseteq 2N(multi)-P^{2k-1}$, and hence (by Theorems 2.6 and 3.5(3)) that $DTIME(exp_{2k-1}(poly)) \subseteq DTIME(exp_{2k-2}(poly))$, contradicting Proposition 7.3. It is not clear whether the inclusion $1N-P^k \subseteq 1N-SA^k$ is optimal.

We wish to make an observation on the strength of our translation technique. Note that in the paragraph above we also proved (for $k=1$) that $1N-SA \not\subseteq 1N-P$, i.e., that $1N-P \subsetneq 1N-SA$. This is of course using a sledge hammer to hit a mosquito: this proper inclusion can easily be proven by the pumping lemma for context-free languages (e.g., $\{a^n b^n c^n \mid n \geqslant 1\}$ is in $1N-SA$). However, we know no other way to prove the proper inclusion $1N-P^k \subsetneq 1N-P^{k+1}$ than the one used in Theorem 7.4. It seems that techniques like pumping lemma's become too complicated for such complicated storage types.

In the remainder of this subsection we consider the iterated checking stack and CSPD automata. First the checking stack. Again we obtain the properness of the hierarchy of 1-way iterated checking stack automata. Moreover, we show that there exist 1-way iterated checking stack languages arbitrarily high in the hierarchy of 1-way iterated pushdown languages.

THEOREM 7.5. *For every* $k \geqslant 1$,

(1) $1N-CSA^k \subsetneq 1N-CSA^{k+1}$,

(2) $1N-CSA^k \subseteq 1N-P^{2k}$,

(3) $1N-CSA^k \not\subseteq 1N-P^{k-1}$, *and even* $1D-CSA^k \not\subseteq 1N-P^{k-1}$.

Proof. (1) can be proven as in Theorem 7.4, using the space complexity characterization of $2N(multi)-CSA^k$ in Theorem 5.2. The inclusion of (2) follows from $1N-CSA^k \subseteq 1N-SA^k \subseteq 1N-P^{2k}$. To show (3), i.e. that there exist $1D-CSA^k$ languages not in $1N-P^{k-1}$, assume to the contrary that $1D-CSA^k \subseteq 1N-P^{k-1}$. Then Theorem 7.1 implies that $2N(multi)-CSA^k \subseteq 2N(multi)-P^{k-1}$. Hence, by Theorems 5.2 and 2.6, $DSPACE(exp_{k-1}(poly)) \subseteq DTIME(exp_{k-2}(poly))$. Since $DTIME(exp_{k-2}(poly)) \subseteq DSPACE(exp_{k-2}(poly))$, this contradicts Proposition 7.3. ∎

Actually, Theorem 7.5(1) is already known in the literature because, as we will show now, the $1N-CSA^k$ hierarchy coincides with the 2GSM hierarchy. The 2GSM hierarchy (shown to be proper in Greibach, 1978c, and in Engelfriet, 1982, in a completely different way) consists of all classes $2GSM^k(REG)$, where $2GSM = 2N(1)T$: the class of 2-way nondeterministic finite transductions, and REG is the class of regular languages. In what follows we will use 2GSM instead of $2N(1)T$, and we will call a $2N(1)$ transducer a 2-way gsm (i.e., 2-way generalized sequential machine).

The next lemma shows that $1N-CSA(X)$ can be expressed in terms of

2GSM and $1N-X$, for every storage type X. It is just a variant of the well-known relationship between checking stack automata and 2-way gsm's (Rajlich, 1972; Kiel, 1975; Greibach, 1978a, b, c).

LEMMA 7.6. *For any storage type* X, $1N-CSA(X)=2GSM(1N-X)$.

Proof. (i) We first show that $2GSM(1N-X)\subseteq 1N-CSA(X)$. Let M be a 1-way nondeterministic X automaton and G a 2-way gsm. We have to construct a 1-way nondeterministic CSA(X) automaton M' such that $L(M')=\tau(G)(L(M))$. To find out whether its input string y belongs to $\tau(G)(L(M))$, M' first guesses an input string x (for M and G) on its checking stack, simultaneously simulating M to check that $x\in L(M)$, and then simulates G to check that $(x, y)\in\tau(G)$. If c_0 is the initial X-configuration of M, then $(\mathbb{c}, c_0)$ is the one of M', where $\mathbb{c}$ is the left endmarker (of G).

First M' simulates M, in pushdown mode. At each moment, the top square of the checking stack contains the current X-configuration of M. If M can read an input symbol σ and execute instruction f on X-storage, then M' can execute push(σ, f). In case M can do a λ-move and execute f, M' can execute an appropriate stay(γ, f). Whenever M accepts the input, M' can decide to execute push(\$, id), to move down to the bottom square of the checking stack, and to start the simulation of G, in reading mode. Note that up to now M' did not read its input. M' simulates G by treating its checking stack as the 2-way input tape of G, and its input tape as the 1-way output tape of G.

(ii) We have to show that $1N\text{-}CSA(X)\subseteq 2GSM(1N-X)$. Let M be a 1-way nondeterministic CSA(X) automaton. We may assume that M does not read any input while in pushdown mode. In fact, at each move it could nondeterministically guess the needed input symbols and store them in the symbol part of the stack square. Then, as soon as it enters reading mode, it could first read these symbols from the input. We may also assume, as observed in the beginning of Section 3, that every test test(t) is false in reading mode.

With these assumptions it is easy to construct a 1-way nondeterministic X automaton M' and a 2-way gsm G such that $\tau(G)(L(M'))=L(M)$. M' just simulates the pushdown mode phase of M. If (γ, c_0) is the initial configuration of M, then M' has initial configuration c_0, and starts by reading γ from its input. When M executes a push(γ, f) instruction, M' reads γ from the input and executes f. Moreover, in order to simulate the sym$=\gamma$ tests of M, M' always keeps the last input symbol in its finite control. When M goes into reading mode, M' first reads its current state (viewed as a symbol) from the input tape, and then accepts. According to this last trick, the 2-way gsm G can start by walking to the right end of its input

and read the state of M, in which it then continues to simulate M's reading mode phase, in the obvious way (note that G does not have to execute test(t) tests). ▮

It now easily follows by induction that the iterated CSA hierarchy and the 2GSM hierarchy are the same.

COROLLARY 7.7. *For every* $k \geq 0$, $1N - CSA^k = 2GSM^k(REG)$.

Note that Theorem 7.5(2, 3) shows that the 2GSM hierarchy is contained in the OI-hierarchy, and that there exist languages in $\bigcup_k 2GSM^k(REG)$ that are arbitrarily high in the OI-hierarchy.

Let K be a trio (i.e., a class of languages closed under intersection with regular languages, inverse homomorphisms, and λ-free homomorphisms; see Ginsburg, 1975). It is well known (see Greibach, 1978c, and p. 122 of Engelfriet, 1982) that either $2GSM^k(K) = 2GSM(K)$ for all $k \geq 1$, or $2GSM^k(K) \subsetneq 2GSM^{k+1}(K)$ for all $k \geq 0$. In other words, either the $2GSM^k(K)$ hierarchy collapses at the first level, or it is proper at each level. In Greibach (1978c) conditions on K are given that guarantee properness of the hierarchy. Here we give another such condition.

THEOREM 7.8. *Let K be a trio such that $K \subseteq 1N - P^n$ for some n. Then $2GSM^k(K) \subsetneq 2GSM^{k+1}(K)$ for every $k \geq 0$.*

Proof. By the discussion above it suffices to show that $2GSM(K) \subsetneq 2GSM^k(K)$ for some k. Thus, since $REG \subseteq K \subseteq 1N - P^n$, it suffices to show that $2GSM^k(REG) \not\subseteq 2GSM(1N - P^n)$ for some k. Now, by Corollary 7.7, $2GSM^k(REG) = 1N - CSA^k$, and, by Lemma 7.6, $2GSM(1N - P^n) = 1N - CSA(P^n)$. Since $CSA(X) \leqslant P(P(X))$ by Lemma 3.1, $1N - CSA(P^n) \subseteq 1N - P^{n+2}$. Thus it suffices to show that $1N - CSA^k \not\subseteq 1N - P^{n+2}$ for some k. This holds for $k = n + 3$ by Theorem 7.5(3). ▮

Since $1N - X$ is a trio for every storage type X, this result holds in particular for $1N - P^n$ itself.

We now turn to iterated CSPD automata. Theorem 7.5 also holds for CSPD instead of CSA, by the space complexity characterization in Theorem 6.3, and by Lemma 6.1. As for CSA automata, the first two results obtained in this way are already known in the literature. As we will discuss next, this is because the 1N-CSPD^k hierarchy is in fact the ETOL hierarchy (see Asveld and van Leeuwen, 1975; Engelfriet, 1982).

ETOL systems are one of the main classes of L-systems, a well-known type of parallel rewriting systems (see Rozenberg and Salomaa, 1980). For a class K of languages, ETOL(K) denotes the class of K-controlled ETOL languages, i.e., languages generated by ETOL systems of which the

derivations are controlled by the strings of a language from K (Asveld, 1977). The ETOL hierarchy is obtained by iterating this mechanism of control on ETOL systems, i.e., it consists of all classes $\mathrm{ETOL}^k(\mathrm{REG})$. It was shown in van Leeuwen (1976) that $\mathrm{ETOL}(\mathrm{REG}) = 1\mathrm{N} - \mathrm{CSPD}$, and, based on this, it was shown in Corollary 4.6 of Engelfriet, Rozenberg, and Slutzki (1980) that, under a few closure conditions on K, $\mathrm{ETOL}(K) = \mathrm{CSPDT}(K)$, where CSPDT denotes the class of transductions realized by cspd transducers, explained next. Thus, the ETOL hierarchy consists of all classes $\mathrm{CSPDT}^k(\mathrm{REG})$.

A *cspd transducer* is a $2\mathrm{N}(1)T-P$ transducer (i.e., a 2-way one-head pushdown transducer) of which the pushdown instructions are coupled to the movements of the input head in the following way. The transducer pushes a symbol on the pushdown when moving its input head to the right, and pops a symbol from the pushdown when moving one square to the left. Thus, the length of the pushdown always equals the number of squares to the left of (and including) the square of the input head. For more details see Engelfriet, Rozenberg, and Slutzki (1980).

It should be clear from this description that a cspd transducer M acts in the same way as a CSPD automaton A in reading mode, viewing the input tape of M as the checking stack of A (with its top to the left) and the output tape of M as the input tape of A. Thus the next lemma should not come as a surprise. Recall that, for a class K of languages, K^{R} denotes the class of all reverses of languages from K.

LEMMA 7.9. *For any storage type* X, $1\mathrm{N} - \mathrm{CSPD}(X) = \mathrm{CSPDT}((1\mathrm{N}-X)^{\mathrm{R}})$.

Proof. The proof is exactly the same as the one of Lemma 7.6, except that, as a moment of thought will reveal, the checking stack of the 1N-CSPD(X) automaton corresponds to the reverse of the input tape of the cspd transducer. ∎

We now show that the iterated CSPD hierarchy is the ETOL hierarchy.

THEOREM 7.10. *For every* $k \geq 0$, $1\mathrm{N} - \mathrm{CSPD}^k = \mathrm{CSPDT}^k(\mathrm{REG})$.

Proof. This follows by induction on k from Lemma 7.9, if we can show that $\mathrm{CSPDT}^k(\mathrm{REG})$ is closed under reversal. In fact, for any class K of languages, $\mathrm{CSPDT}(K)$ is closed under reversal. Let M be a cspd transducer. Then a cspd transducer M' can be constructed such that $\tau(M') = \{(x, y^{\mathrm{R}}) \mid (x, y) \in \tau(M)\}$. We may assume that M accepts in a unique final state, with its input head on the left endmarker $\mathfrak{c}$. M' simulates a computation of M in reverse. Thus, M' starts in the final state of M, at $\mathfrak{c}$. M' then simulates the moves of M backwards until it arrives at $\mathfrak{c}$ in an

accepting state of M. As an example, suppose that M, in state q, scanning σ on its input tape and γ on the top of its pushdown, can go into state q', and move its input head to the right, pushing γ'. Then M', in state q' and scanning γ' on top of its pushdown, can go into state q and move its input head to the left, after which it should check that it scans σ on its input tape and γ on top of its pushdown. In case M moves to the left, popping the pushdown, M' moves to the right, guessing a symbol to push on the pushdown. ∎

Properness of the ETOL hierarchy (which was proved in Engelfriet, 1982, by other means) now follows from properness of the hierarchy 1N-CSPD^k of iterated CSPD automata. In fact, since 2GSM^k(REG) $\not\subseteq$ CSPDT^{k-1}(REG) by Theorems 5.2 and 6.3, the counterexamples are already in the 2GSM hierarchy (as also shown in Theorem 4.7 of Engelfriet, 1982). The inclusion $1N-\text{CSPD}^k \subseteq 1N-P^{2k}$ proves that the ETOL hierarchy is contained in the OI-hierarchy (proven in Vogler, 1988, by showing that the OI-hierarchy is closed under control). It can be shown that the inclusion is proper for every k. This is well known for $k=1$, and can be shown in a way similar to the proof of Theorem 7.5(3) for $k \geqslant 2$, using the fact that $\text{DSPACE}(\exp_{k-1}(\text{poly})) \subsetneq \text{DTIME}(\exp_{2k-1}(\text{poly}))$ for $k \geqslant 2$. However, it is open whether $\bigcup_k 1N-\text{CSPD}^k \subsetneq \bigcup_k 1N-P^k$, i.e., whether the ETOL hierarchy is properly contained in the OI-hierarchy. Since it is shown in (Greibach, 1978c) that there is a context-free language not in $\bigcup_k$ 2GSM^k(REG), the 2GSM hierarchy is properly contained in the ETOL hierarchy (because $1N-P \subseteq 1N-\text{CSPD}$).

Thus, we have shown that the seemingly unrelated 2GSM hierarchy, ETOL hierarchy, and OI-hierarchy can be described in a uniform way by iterated X automata, where X is CSA, CSPD, and P (or P^2), respectively.

7.2. *The Emptiness Problem for One-Way Iterated Automata*

It is well known, for several storage types X, that the nonemptiness problem for one-way X automata is complete in the (complexity) class of languages accepted by the multi-head X automata (see, e.g., Jones, 1975, Jones and Laaser, 1977, Galil, 1977, and Hunt, 1976). In the next result we show that this is a general phenomenon. The proof is similar to those in the above references. A storage type $X=(C, T, F, m, C_0, \text{id})$ is *finitely encoded* (see Chap. 5 of Ginsburg, 1975) if T and F are finite.

THEOREM 7.11. *Let X be a finitely encoded storage type. The nonemptiness problem for one-way nondeterministic X automata is log-space complete in* $2N(\text{multi})-X$.

Proof. Let $A(X)$ denote the class of one-way nondeterministic X automata, and also, ambiguously, the set of strings that code these

automata over a fixed alphabet, in the usual way. In particular, the transition function of an automaton M in $A(X)$ is specified as a sequence of tuples in

$$Q \times (\Sigma \cup \{\lambda\}) \times R(T_M) \times Q \times F_M$$

and each such tuple is called a transition of M.

We first construct a $2N(2)-X$ automaton N that accepts the language $\{M \in A(X) \mid L(M) \neq \varnothing\}$. Given an input string M, N first checks that $M \in A(X)$, and then simulates M, guessing nondeterministically some input string for M that hopefully is accepted by M. N uses one head to point to the current transition of M, simulates the execution of this transition (which is possible because X is finitely encoded), and uses the second head to find a new transition. The correct state behaviour of M is guaranteed by N checking equality of the (coded) states in the old and the new transition, using both heads.

Next, let N be a $2N(r)-X$ automaton for some $r \geqslant 1$. We have to show that $L(N)$ is log-space reducible to the nonemptiness problem $\{M \in A(X) \mid L(M) \neq \varnothing\}$. Let w be a given input string of N, of length n. We construct $N_w \in A(X)$ such that, independent of its input, N_w simulates N on w by keeping track of N's head positions on w in its finite control. Thus, N_w has states $\langle q, i_1, ..., i_r \rangle$ where q is a state of N and $0 \leqslant i_j \leqslant n+1$ for every $1 \leqslant j \leqslant r$. Clearly $w \in L(N)$ if and only if $L(N_w) \neq \varnothing$. The translation from w into N_w can be realized by a DSPACE($\log n$) transducer. In fact, N_w contains for each $\langle i_1, ..., i_r \rangle$ a set of transitions that is easily obtainable from the transitions of N. Thus, N_w has $O(n^r)$ transitions. To write down a sequence $\langle i_1, ..., i_r \rangle$ takes $\log n$ space, and it suffices to keep track of these sequences. Note that N_w is of length $O(n^r \log n)$. ∎

We note that this theorem also holds if the one-way X automata are restricted to be deterministic: in this case N_w should use its own input string to decide which transitions of N to choose. The theorem also holds for the "general membership problem," i.e., the language $\{(M, x) \mid M \in A(X), x \in L(M)\}$. In fact, on the one hand, this language can easily be accepted by a $2N(\text{multi})-X$ automaton. On the other hand, w can be translated into (N_w, λ), because $w \in L(N)$ iff $\lambda \in L(N_w)$. Finally, it also holds for every nontrivial property of $1N-X$ languages that can be decided by a $2N(\text{multi})-X$ automaton (cf. Hunt, 1976): after successful simulation of N on w, N_w should simulate (on its own input) some $1N-X$ automaton M such that $L(M)$ has or does not have the property, depending on whether $\varnothing$ does not have or has the property.

To be able to use Theorem 7.11 we will assume in the remainder of this section (just as in Hunt, 1976) that the set Γ of pushdown or stack symbols is finite, or even that $\Gamma = \{0, 1\}$. It is easy to see that, for every storage type

considered in this paper, this restriction results in an equivalent storage type (that will be denoted by the same name). This was discussed in Section 1.3 for the storage type $P(X)$. Of course, the restriction gives a smaller class of one-way automata.

With this restriction on Γ, all our storage types are finitely encoded. Thus, we immediately obtain from Theorem 7.11 (for $X = P^k$) and from the complexity characterization of $2N(multi) - P^k$ in Theorem 2.6, that for $k \geqslant 1$ the emptiness problem for $1N - P^k$ automata is log-space complete in $DTIME(\exp_{k-1}(poly))$, and that the same holds for $1D - P^k$ automata. This result can be sharpened as follows.

THEOREM 7.12. *For $k \geqslant 2$,*

(1) *the emptiness problem for $1N - P^k$ automata is in $\bigcup DTIME(\exp_{k-1}(dn^2))$, and*

(2) *for every $\varepsilon > 0$, the emptiness problem for $1D - P^k$ automata is not in $\bigcup DTIME(\exp_{k-1}(dn^{2-\varepsilon}))$.*

Proof. (1) This nonemptiness problem can be accepted by a $2N(multi) - P^k$ automaton with one head rather than two, cf. Proposition 4.5 of Hunt (1976). In fact (referring to the proof of Theorem 7.11), to check equality of the old and the new state, N stores the (coded) old state on its "outer" pushdown, by appropriate push(γ, id) instructions, and then compares it to the new state, popping the pushdown. The result now follows from Theorem 3.3 with $r = 1$.

(2) The second part of the proof of Theorem 7.11 shows that the log-space reduction of languages in $2N(1) - X$ produces strings of length $O(n \log n)$. The result now follows from the fact that, by the time hierarchy theorem, $\bigcup_{d>0} DTIME(\exp_{k-1}(dn^{2-\varepsilon}(\log n)^{2-\varepsilon}))$ is properly contained in $DTIME(\exp_{k-1}(dn^2))$ for every d. ∎

Decidability of this problem was first shown in (Damm, 1982), using algebraic methods, for the k-level OI macro grammars. It is mentioned in (Greibach, 1970) as having been shown by Aho and Ullman.

Theorem 7.12 gives some natural automata-theoretic decision problems of high intractability. By combining them, a nonelementary problem is obtained; cf. the result of Meyer and Stockmeyer in Section 11.4 of (Aho, Hopcroft, and Ullman, 1974). Recall that $\bigcup_k DTIME(\exp_k(poly))$ is the class of elementary problems. Let an iterated pushdown automaton be a P^k automaton for any k (where the k is specified in the automaton).

COROLLARY 7.13. *The emptiness problem for one-way (non)deterministic iterated pushdown automata is decidable, but not elementary.*

Proof. Decidability follows from the fact that the appropriate multi-head P^k automaton can be computed from k, and from the effectiveness of all our results (in particular Theorem 2.6). ▮

It can be shown that Theorem 7.12 and Corollary 7.13 also hold for infinite Γ. For the lower bounds this is of course trivial. For the upper bounds one has to implement the binary encoding of the stack symbols, in the obvious way.

It should be clear that similar results can be obtained for all other storage types considered. For iterated checking stack automata this implies the following result. By $2GSM^k$ we denote the class of all relations that are compositions of k 2-way gsm's. The emptiness problem for this class is the problem of deciding for given 2-way gsm's $M_1, ..., M_k$ whether the composition of $\tau(M_1)$ to $\tau(M_k)$ is the empty relation.

THEOREM 7.14. *For every* $k \geqslant 1$, *the emptiness problem for* $2GSM^k$ *is log-space complete in* $DSPACE(exp_{k-1}(poly))$.

Proof. From Theorems 7.11 and 5.2 we obtain completeness of the emptiness problem for $1N-CSA^k$ automata in $DSPACE(exp_{k-1}(poly))$. It can be checked that all translations in Lemma 7.6 (and hence in Corollary 7.7) are $DSPACE(\log n)$ transductions (including a translation that changes the $CSA(X)$ automaton into one that uses stack symbols 0 and 1 only). This shows that the emptiness problem for $2GSM^k(REG)$ is complete in $DSPACE(exp_{k-1}(poly))$. Clearly, incorporating the regular language into the finite control of the first 2-way gsm, the same holds for the ranges of the relations in $2GSM^k$, and hence for the relations themselves. ▮

Since $\bigcup_k DSPACE(exp_k(poly))$ is the class of elementary problems, we obtain the following intractable problem for 2-way nondeterministic finite state transducers.

THEOREM 7.15. *The emptiness problem for compositions of* 2-way gsm's *is decidable, but not elementary.*

CONCLUSION

Some remaining questions are the following.

(1) What is the power (in terms of complexity classes) of deterministic r-head P^k automata?

(2) What is the precise power of alternating multi-head CSA^k automata? It is straightforward to show that $\bigcup ASPACE(exp_1(ds(n)))-$

$X \subseteq \text{ASPACE}(s(n)) - \text{CSA}(X) \subseteq \bigcup \text{ASPACE}(\exp_2(ds(n))) - X$. If the $\text{CSA}(X)$ automaton is not allowed to branch universally in reading mode, $\bigcup \text{ASPACE}(\exp_1(ds(n))) - X$ is obtained. For CSPD automata it is not difficult to show that $\text{ASPACE}(s(n)) - \text{CSPD}(X) = \bigcup \text{ASPACE}(\exp_2(ds(n))) - X$.

(3) Are the $1\text{N} - P^k$ languages context-sensitive? This is mentioned in (Greibach, 1970) as having been shown by Aho and Ullman.

(4) Questions like: is $1\text{N} - \text{SA}^k$ properly included in $1\text{N} - P^{2k}$?

(5) Is it possible to find automaton-theoretic characterizations of complexity classes larger than the class of elementary languages? One idea is to define an iterated pushdown automaton with an arbitrary number of levels of pushdowns: it should have instructions to go up and down in level. Unfortunately, this automaton would be able to accept all recursively enumerable languages (each level would simulate one square of a Turing machine tape). One could think of bounding the number of levels in terms of the length of the input.

(6) In general, which complexity classes can be characterized by 2-way or multi-head automata (in particular NTIME classes), and, vice versa, for which storage types X are $2\text{N}(1) - X$ and $2\text{N}(\text{multi}) - X$ complexity classes?

ACKNOWLEDGMENT

I thank Werner Damm for stimulating discussions.

RECEIVED June 12, 1989; FINAL MANUSCRIPT RECEIVED April 6, 1990

REFERENCES

AHO, A. V. (1968), Indexed grammars, an extension of context-free grammars, *J. Assoc. Comput. Mach.* **15**, 647–671.

AHO, A. V. (1969), Nested stack automata, *J. Assoc. Comput. Mach.* **16**, 383–406.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Ma.

ASVELD, P. R. J. (1977), Controlled iteration grammars and full hyper-AFL's, *Inform. and Control* **34**, 248–269.

ASVELD, P. R. J., AND VAN LEEUWEN, J. (1975), "Infinite Chains of Hyper-AFL's," Memorandum 99, Twente University of Technology.

BEERI, C. (1975), Two-way nested stack automata are equivalent to two-way stack automata, *J. Comput. System Sci.* **10**, 317–339.

CHANDRA, A. K., KOZEN, D. C., AND STOCKMEYER, L. J. (1981), Alternation, *J. Assoc. Comput. Mach.* **28**, 114–133.

COOK, S. A. (1971), Characterizations of pushdown machines in terms of time-bounded computers, *J. Assoc. Comput. Mach.* **18**, 4–18.

DAMM, W. (1982), The IO- and OI-hierarchies, *Theoret. Comput. Sci.* **20**, 95–206.

DAMM, W., AND GOERDT, A. (1986), An automata-theoretical characterization of the OI-hierarchy, *Inform. and Control* **71**, 1–32.

ENGELFRIET, J. (1982), Three hierarchies of transducers, *Math. Systems Theory* **15**, 95–125.

ENGELFRIET, J. (1983), Iterated pushdown automata and complexity classes, *in* "Proceedings, 15th STOC, Boston," pp. 365–373.

ENGELFRIET, J. (1986), "Context-Free Grammars with Storage," Report 86-11, Leiden University.

ENGELFRIET, J., AND HOOGEBOOM, H. J. (1989), "X-Automata on ω-words," Report 89-06, Leiden University. See also "Proceedings 16th ICALP, Lecture Notes in Computer Science," Vol. 372, Springer-Verlag, Berlin, 1989, 289–303.

ENGELFRIET, J., ROZENBERG, G., AND SLUTZKI, G. (1980), Tree transducers, L systems, and two-way machines, *J. Comput. System Sci.* **20**, 150–202.

ENGELFRIET, J., AND SCHMIDT, E. M. (1977/1978), IO and OI, I, *J. Comput. System Sci.* **15**, 328–353; II, *J. Comput. System Sci.* **16**, 67–99.

ENGELFRIET, J., SCHMIDT, E. M., AND VAN LEEUWEN, J. (1980), Stack machines and classes of nonnested macro languages, *J. Assoc. Comput. Mach.* **27**, 96–117.

ENGELFRIET, J., AND SLUTZKI, G. (1984), Extended macro grammars and stack controlled machines, *J. Comput. System Sci.* **29**, 366–408.

ENGELFRIET, J., AND VOGLER, H. (1986), Pushdown machines for the macro tree transducer, *Theoret. Comput. Sci.* **42**, 251–368.

ENGELFRIET, J., AND VOGLER, H. (1987), Look-ahead on pushdowns, *Inform. and Comput.* **73**, 245–279.

ENGELFRIET, J., AND VOGLER, H. (1988), High level tree transducers and iterated pushdown tree transducers, *Acta Inform.* **26**, 131–192.

FISCHER, M. J. (1969), Two characterizations of the context-sensitive languages, *in* "IEEE Conference Record of 10th Annual Symposium on Switching and Automata Theory," pp. 149–165.

GALIL, Z. (1977), Hierarchies of complete problems, *Acta Inform.* **6**, 77–88.

GINSBURG, S. (1975), "Algebraic and Automata-Theoretic Properties of Formal Languages," North-Holland, Amsterdam.

GINSBURG, S., GREIBACH, S. A., AND HARRISON, M. A. (1967), Stack automata and compiling, *J. Assoc. Comput. Mach.* **14**, 389–418.

GOLDSTINE, J. (1977), Automata with data storage, *in* "Proceedings of a Conference on Theoretical Computer Science, Waterloo," pp. 239–246.

GREIBACH, S. A. (1969), Checking automata and one-way stack languages, *J. Comput. System Sci.* **3**, 196–217.

GREIBACH, S. A. (1970), Full AFLs and nested iterated substitution, *Inform. and Control* **16**, 7–35.

GREIBACH, S. A. (1978a), Visits, crosses, and reversals for nondeterministic off-line machines, *Inform. and Control* **36**, 174–216.

GREIBACH, S. A. (1978b), One-way finite visit automata, *Theoret. Comput. Sci.* **6**, 175–221.

GREIBACH, S. A. (1978c), Hierarchy theorems for two-way finite state transducers, *Acta Inform.* **11**, 89–101.

HOARE, C. A. R. (1972), Proof of correctness of data representations, *Acta Inform.* **1**, 271–281.

HOPCROFT, J. E., AND ULLMAN, J. D. (1967a), An approach to a unified theory of automata, *Bell System Tech. J.* **46**, 1793–1829.

HOPCROFT, J. E., AND ULLMAN, J. D. (1967b), Nonerasing stack automata, *J. Comput. System Sci.* **1**, 166–186.

HOPCROFT, J. E., AND ULLMAN, J. D. (1979), "Introduction to Automata Theory, Languages, and Computation," Addison–Wesley, Reading, MA.

HUNT III, H. B. (1976), On the complexity of finite, pushdown, and stack automata, *Math. Systems Theory* **10**, 33–52.

IBARRA, O. H. (1971), Characterizations of some tape and time complexity classes of Turing machines in terms of multi-head and auxiliary stack automata, *J. Comput. System Sci.* **5**, 88–117.

JONES, N. D. (1975), Space-bounded reducibility among combinatorial problems, *J. Comput. System Sci.* **11**, 68–85.

JONES, N. D., AND LAASER, W. T. (1977), Complete problems for deterministic polynomial time, *Theoret. Comput. Sci.* **3**, 105–117.

KIEL, D. I. (1975), Two-way a-transducers and AFL, *J. Comput. System Sci.* **10**, 88–109.

KOWALCZYK, W., NIWINSKI, D., AND TIURYN, J. (1989), A generalization of Cook's auxiliary-pushdown-automata theorem, *Fund. Inform.* **12**, 497–506.

LADNER, R. E., LIPTON, R. J., AND STOCKMEYER, L. J. (1984), Alternating pushdown and stack automata, *SIAM J. Comput.* **13**, 135–155.

MAIBAUM, T. S. E. (1974), A generalized approach to formal languages, *J. Comput. System Sci.* **8**, 409–439.

MASLOV, A. N. (1974), The hierarchy of indexed languages of an arbitrary level, *Soviet Math. Dokl.* **15**, 1170–1174.

MASLOV, A. N. (1976), Multi-level stack automata, *Problems Inform. Transmission* **12**, 38–43.

PARCHMANN, R., DUSKE, J., AND SPECHT, J. (1980), On deterministic indexed languages, *Inform. and Control* **45**, 48–67.

RAJLICH, V. (1972), Absolutely parallel grammars and two-way finite state transducers, *J. Comput. System Sci.* **6**, 324–342.

ROZENBERG, G., AND SALOMAA, A. (1980), "The Mathematical Theory of *L* Systems," Academic Press, New York.

RUZZO, W. L. (1980), Tree-size bounded alternation, *J. Comput. System Sci.* **21**, 218–235.

SCOTT, D. (1967), Some definitional suggestions for automata theory, *J. Comput. System Sci.* **1**, 187–212.

VAN LEEUWEN, J. (1976), Variations of a new machine model, *in* "Proceedings, 17th FOCS."

VOGEL, J., AND WAGNER, K. (1985), Two-way automata with more than one storage medium, *Theoret. Comput. Sci.* **39**, 267–280.

VOGLER, H. (1986), Iterated linear control and iterated one-turn pushdowns, *Math. Syst. Theory* **19**, 117–133.

VOGLER, H. (1988), The OI-hierarchy is closed under control, *Inform. and Comput.* **78**, 187–204.

WAGNER, K., AND WECHSUNG, G. (1986), "Computational Complexity," Reidel, Dordrecht.

WAND, M. (1975), An algebraic formulation of the Chomsky-hierarchy, *in* "Category Theory Applied to Computation and Control," Lecture Notes in Computer Sci., Vol. 25, Springer-Verlag, Berlin, pp. 209–213.