

Extended Macro Grammars and Stack Controlled Machines

JOOST ENGELFRIET AND GIORA SLUTZKI*

*Department of Applied Mathematics, Twente University of Technology,
P.O. Box 217, 7500AE Enschede, The Netherlands*

Received July 26, 1983; revised July 2, 1984

K -extended basic macro grammars are introduced, where K is any class of languages. The class $B(K)$ of languages generated by such grammars is investigated, together with the class $LB(K)$ of languages generated by the corresponding linear basic grammars. For any full semi-AFL K , $B(K)$ is a full AFL closed under iterated $LB(K)$ -substitution, but not necessarily under substitution. For any machine type D , the stack controlled machine type corresponding to D is introduced, denoted $S(D)$, and the checking-stack controlled machine type $CS(D)$. The data structure of this machine is a stack which controls a pushdown of data structures from D . If D accepts K , then $S(D)$ accepts $B(K)$ and $CS(D)$ accepts $LB(K)$. Thus the classes $B(K)$ are characterized by stack controlled machines and the classes $LB(K)$, i.e., the full hyper-AFLs, by checking-stack controlled machines. A full basic-AFL is a full AFL K such that $B(K) \subseteq K$. Every full basic-AFL is a full hyper-AFL, but not vice versa. The class of OI macro languages (i.e., indexed languages, i.e., nested stack automaton languages) is a full basic-AFL, properly containing the smallest full basic-AFL. The latter is generated by the ultrabasic macro grammars and accepted by the nested stack automata with bounded depth of nesting (and properly contains the stack languages, the ETOL languages, i.e., the smallest full hyper-AFL, and the basic macro languages). The full basic-AFLs are characterized by bounded nested stack controlled machines. © 1984 Academic Press, Inc.

INTRODUCTION

One of the nicest aspects of AFL theory [20] is that it reveals a close connection between the closure properties of a class of languages and the properties of the class of accepting automata by which it is defined. The main result is that a class of languages is a full AFL if and only if it is defined by an "abstract family of acceptors" (AFA). Similarly, full substitution-closed AFLs are characterized by nested multitape AFAs, and (full) super-AFLs, i.e., full AFLs closed under iterated nested substitution, correspond to nested AFAs [26]. Thus, in general, full AFLs with additional closure properties are characterized by classes of machines with an additional structure on their memory. (We use the words machine, automaton, and acceptor synonymously in this paper).

* Supported by the National Science Foundation under Grant MCS-8024341

In this paper we establish a machine characterization for the full hyper-AFLs [36, 34, 4] which are full AFLs closed under iterated substitution (for a different characterization, see [33]). Moreover, we define an operation more powerful than iterated substitution (which we call basic substitution) and provide a machine characterization for the full AFLs closed under that operation (which we call full basic-AFLs).

The motivation to introduce this rather artificial new operation of basic substitution is that it can be used successfully as a technical tool to show the existence of an infinite hierarchy of full hyper-AFLs inside the class of indexed languages: this will be shown in the companion paper [13] (note that until now the only known fact about the hyper-AFL structure of the class of indexed languages is that it is a full hyper-AFL properly containing the smallest full hyper-AFL ETOL, see [10, 15]). In fact, it turns out in [13] that basic substitution is even a quite natural technical tool in the study of full hyper-AFLs: basic substitution plays the same role for full hyper-AFLs as ordinary substitution does for full AFLs (in the sense of [28]). Motivated by this success as a technical tool we take the courage to introduce basic substitution in this paper, to study its main closure properties (most of which are needed in [13]), and to provide a machine characterization of full basic-AFLs, closely related to the one of full hyper-AFLs. It turns out that the smallest full basic-AFL is a very natural class of languages inside the class of indexed languages: it is characterized by macro grammars with bounded nesting of nonterminals (the ultra-basic macro grammars of [16]) and by nested stack automata with bounded nesting of stacks. Using a result of [13], it follows that the nested stack automata form an infinite hierarchy with respect to depth of nesting of stacks. This is the most significant concrete result that seems impossible to prove without the concepts introduced in this paper. We observe that although many of our results are given in a general framework, the ultimate aim of these investigations is to establish the formal language theoretic properties of the indexed languages.

Basic substitution will be defined in terms of “extended” macro grammars. To explain how this can be done we first have to recall how operations on languages can be defined by way of “extended grammars” (for a longer discussion, see Chapter 1 of [5]). As a first example, the generating process of a context free grammar can be viewed as the iteration of a nested finite substitution. Generalizing this idea, the operation of iterated nested substitution on an arbitrary class K of languages can be defined by a K -extended context-free grammar, which is a context-free grammar with an infinite number of rules such that the set of right-hand sides of rules for a given nonterminal form a language from K (see [37]). If $CF(K)$ denotes the class of languages generated by all K -extended context-free grammars, then K is a full super-AFL if and only if K is a full AFL and $CF(K) \subseteq K$. Similarly, the parallel generation process of an ETOL system (which is one of the main types of L -systems [32]) can be viewed as an iterated finite substitution (not necessarily nested) and, generalizing this idea as for the context-free case, the notion of K -extended ETOL system (or, K -iteration grammar) is obtained ([36], see also [34, 4]). Again, if

$H(K)$ denotes the class of languages generated by arbitrary K -iteration grammars, then a full hyper-AFL is a full AFL K such that $H(K) \subseteq K$. In fact, under appropriate closure properties of K , $CF(K)$, and $H(K)$ are the smallest full super-AFL and full hyper-AFL containing K , respectively. We now arrive at the link with macro grammars [18]. It was discovered in [9, 3] that deterministic ETOL systems (which correspond to the iteration of a finite set of homomorphisms) are equivalent in generating power to the linear basic macro grammars (i.e., macro grammars with at most one nonterminal in the right-hand side of each rule). Moreover, in [9], the notion of a (finitely) extended linear basic macro grammar was proposed, which is essentially a linear basic macro grammar with a finite set of strings rather than one string in each argument of each nonterminal; it was proved there that these grammars are equivalent to ETOL systems. This showed that iterated finite substitution is closely related to macro grammars. Then, in [7], K -extended linear basic macro grammars were introduced, i.e., linear basic macro grammars in which the arguments of the nonterminals may hold arbitrary languages from K . Denoting by $LB(K)$ the class of languages generated by such grammars, it was shown (under weak assumptions on K) that $LB(K) = H(K)$, and so iterated substitution can be characterized by extended linear basic macro grammars in general. The operation introduced in this paper is obtained by generalizing this idea to the basic macro grammars (i.e., macro grammars with no nesting of nonterminals in the right-hand side of rules): K -extended basic macro grammars. Each K -extended basic macro grammar can be viewed as a "basic substitution" on K . We study the properties of the class $B(K)$ of languages generated by K -extended basic macro grammars and investigate full basic AFLs, i.e., full AFLs K such that $B(K) \subseteq K$. One of the main examples of a full basic-AFL is the class OI of languages generated by all macro grammars (alternatively [18], it is the class of indexed languages [1] and the class of nested stack automaton languages [2]). We will show in particular that $B(K)$ is not always the smallest full basic-AFL containing K (as was the case for $CF(K)$ and $H(K) = LB(K)$). In fact, $B(\text{FIN})$ is not even substitution-closed (where FIN denotes the class of finite languages).

For $K = \text{FIN}$, the class $B(K)$ has already been introduced (under the name EB, extended basic) in [15], where it was shown that EB corresponds to the class of s-pd machines (stack pushdown machines) which are a slight generalization of the usual stack automata [22]. Similarly, the class ELB ($= LB(\text{FIN}) = \text{ETOL}$) can be defined by the cs-pd machines (checking-stack pushdown machines) which generalize in the same way the checking-stack automata [27]. Cs-pd machines were introduced, as acceptors of ETOL, in [38]. In this paper we define the notions of checking-stack controlled AFA and stack-controlled AFA, and show that these types of machines characterize the classes $LB(K)$, i.e., the full hyper-AFLs, and the classes $B(K)$, respectively (where K has some appropriate closure properties). If D is an AFA for the class K , then the data structure of the stack controlled AFA corresponding to D (denoted $S(D)$) is obtained by replacing, in the data structure of the s-pd machine (i.e., a stack together with a pushdown whose bottom is at the top of the stack and whose top is at the stack pointer), each pushdown square by a

tape which is a data structure of D . Thus, it may be viewed as a stack controlled pushdown of D -tapes (it generalizes the nested AFA, corresponding to the super-AFL, which consists of just a pushdown of D -tapes [26]). In the same way $CS(D)$, the checking-stack controlled AFA corresponding to D , has a checking-stack with a pushdown of D -tapes as data structure (generalizing the cs-pd machine). A machine characterization of the full basic-AFLs can be obtained by iterating the operation $S(D)$: thus AFAs of the form $\bigcup\{S^k(D)|k \geq 1\}$ characterize full basic-AFLs. We will show that these AFA correspond to nested stack automata [2] with a fixed bound on the depth of nesting of stacks. In particular, the smallest full basic-AFL is the class of languages accepted by such bounded nested stack automata (properly contained in OI).

It should be clear to the reader by now that this paper is a generalization of two other papers, viz. [26 and 15]. In fact, this paper is related to the s-pd machine of [15] in exactly the same way as [26] is related to the pushdown automaton.

This paper is divided into six sections. The first section contains preliminary definitions and notation (in particular, concerning iteration grammars, macro grammars, and AFL and AFA theory. Section 2 gives the definition of K -extended basic macro grammars and some of their elementary properties.

In Section 3 closure properties of the classes $B(K)$ and $LB(K)$ are investigated (most of which are needed in [13]). Under appropriate conditions on K , $B(K)$ is a full AFL closed under iterated $LB(K)$ -substitution (and $LB(K)$ is a full hyper-AFL). $LB(K)$ is the largest substitution-closed class of languages inside $B(K)$. The smallest full basic-AFL $B^*(\text{FIN})$ equals the class of ultrabasic macro languages [16]. Hence $\text{ETOL} \subsetneq \text{EB} \subsetneq B^*(\text{FIN}) \subsetneq \text{OI}$. This shows the existence of a full hyper-AFL, viz. $B^*(\text{FIN})$, properly between the smallest full hyper-AFL ETOL and the class OI of indexed languages. It also shows that OI is not reachable from the stack-pushdown languages (i.e., $B(\text{FIN})$) by full hyper-AFL operations, strengthening the result of [26] that OI is not reachable from the stack languages by full super-AFL operations. These results will be further strengthened in [13].

In Section 4 we define the (checking) stack controlled machine types $S(D)$, and $CS(D)$, where D is any given (nontrivial) machine type, and show that if D accepts the class K of languages, then $S(D)$ accepts $B(K)$ and $CS(D)$ accepts $LB(K)$. Hence the checking-stack controlled machine types $CS(D)$ characterize the full hyper-AFLs. The new machine types have only finitely many instructions in addition to those of D ; hence, if K is a full principal semi-AFL, then $LB(K)$ and $B(K)$ are also full principal. In [13] it is shown that $B^*(\text{FIN})$ is the union of the infinite hierarchy of full hyper-AFLs $LB(B^n(\text{FIN}))$, $n \geq 0$. The machine characterizations of this section provide concrete machine models for these concrete full hyper-AFLs, viz. $CS(S^n(D_0))$ (where D_0 is the "trivial" machine type): a checking-stack controlled iteration of stack controlled machines.

Finally, in Section 5, we introduce the nested-stack controlled machine types and restrict them to have a bounded depth of nesting of stacks. These machine types, denoted $\text{BNS}(D)$, characterize the full basic-AFLs (intuitively, nesting of stacks up to a bounded depth corresponds to iterations of the $S(D)$ -operation). In fact, if D

accepts K , then $BNS(D)$ accepts $B^*(K) = \bigcup \{B^n(K) | n \geq 1\}$: the smallest full basic-AFL containing K . In particular, the smallest full basic-AFL $B^*(FIN)$ is the class of languages accepted by nested stack automata with bounded depth of nesting. With respect to depth of nesting these automata form an infinite hierarchy (using a result of [13]).

Some of the results of this paper were announced in [12].

Apologies (to one of the referees, and to the reader). We did not take $B(K)$ and basic-AFLs serious at first, calling them $HH(K)$ and hyphyper-AFLs, respectively. They seemed to be generalizations for the sake of generalization. However, they turned out to be more and more useful and natural. Perhaps they are ugly creatures with a beautiful nature.

1. PRELIMINARIES

In this section we list some concepts and notation needed in the rest of the paper.

The empty string is denoted by λ and the empty set by \emptyset . FIN, REG, and CF denote the classes of finite, regular, and context-free languages, respectively.

Let K be a class of languages, and Δ an alphabet. A K -substitution on Δ is a mapping $f: \Delta \rightarrow K$, extended to strings and languages in the usual way: for strings u and v , $f(uv) = f(u) \cdot f(v)$; $f(\lambda) = \{\lambda\}$; for a language L , $f(L) = \bigcup \{f(u) | u \in L\}$. Thus, for $L \subseteq \Delta^*$, $f(L)$ is a language, not necessarily in K . For languages L_0, L_1, \dots, L_n and symbols a_1, \dots, a_n , we use $L_0[a_1 \leftarrow L_1, a_2 \leftarrow L_2, \dots, a_n \leftarrow L_n]$ to denote $f(L_0)$, where f is the substitution such that $f(a_i) = L_i$ and $f(b) = \{b\}$ for $b \neq a_i$. Similarly, if $L(a)$ denotes a language for every $a \in \Sigma$, we write $L[a \leftarrow L(a)]$ for $f(L)$, where $f(a) = L(a)$ for $a \in \Sigma$ and $f(b) = \{b\}$ for $b \notin \Sigma$.

Let K_1, K_2 , and K be classes of languages. Then $K_1 \leftarrow K_2$ denotes the class $\{f(L) | L \in K_1, f \text{ is a } K_2\text{-substitution}\}$. We say that K_1 is closed under K_2 -substitution if $K_1 \leftarrow K_2 \subseteq K_1$, and that K is closed under substitution if it is closed under K -substitution.

We now consider (a slight variation of) iteration grammars, cf. [36, 34, 4, 32]. Let K and K_0 be classes of languages. A K -iteration grammar with axiom set from K_0 is a construct $G = (V, \Sigma, U, A)$, where V is an alphabet, $\Sigma \subseteq V$ is the terminal alphabet, U is a finite set of K -substitutions on V (such that $f(a) \subseteq V^*$ for $f \in U$ and $a \in V$), and $A \subseteq V^*$ is the axiom set with $A \in K_0$. The language generated by G is $L(G) = U^*(A) \cap \Sigma^*$, where $U^*(A) = \bigcup \{f_1(f_2(\dots f_n(A)\dots)) | n \geq 0, f_i \in U\}$. The class of languages generated by all K -iteration grammars with axiom set from K_0 is denoted $H(K_0, K)$ (not to be confused with the notation in [4], where K_0 denotes the class of control languages). We denote $H(K, K)$ also by $H(K)$. $H(FIN)$ is the class of ETOL languages [32]. We say that K_0 is closed under iterated K -substitution if $H(K_0, K) \subseteq K_0$, and that K is closed under iterated substitution if $H(K) \subseteq K$.

Next we discuss other closure properties. We first need the concept of a -transduction: it is the translation realized by an a -transducer, i.e., a one-way finite state

transducer, see [20]. An *ngsm* (nondeterministic generalized sequential machine) is an *a*-transducer that reads exactly one input symbol with each move.

Let K be a class of languages. K is a *prequasoid* [4] if it contains FIN and is closed under FIN-substitution and intersection with regular languages (equivalently, it is closed under *ngsm* mappings). FIN is the smallest prequasoid; all other prequasoids contain REG. Note that every full trio [20] is a prequasoid (a full trio is closed under *a*-transductions). K is a *full semi-AFL* [20, 8] if it contains a non-empty language and is closed under *a*-transductions and union. Every full semi-AFL is a full trio and hence a prequasoid. A full semi-AFL K is *full principal* if, for some $L \in K$, K is the smallest full semi-AFL containing L ; L is called a *generator* of K . A *full AFL* is a full semi-AFL closed under concatenation and Kleene star. A (full) *super-AFL* [26] is a full AFL closed under iterated nested substitution (a substitution f is nested if $a \in f(a)$ for all symbols a). K is a *full hyper-AFL* [36, 34, 4] if it is a full AFL closed under iterated substitution, i.e., $H(K) \subseteq K$.

We surely need the concept of macro grammar ([18], see also [14]). A *ranked alphabet* Δ is a finite set of symbols such that with each symbol $A \in \Delta$ a unique non-negative integer (the rank of A) is associated. For $i \geq 0$, Δ_i denotes the set of all symbols of rank i in Δ . Let PC be the alphabet consisting of the left parenthesis, the right parenthesis, and the comma symbol. The set of (macro) *terms* over Δ is the smallest set of strings over $\Delta \cup PC$ such that: (i) each element of $\Delta_0 \cup \{\lambda\}$ is a term; (ii) if t_1 and t_2 are terms, then $t_1 t_2$ is a term; (iii) if $A \in \Delta_m$ and t_1, \dots, t_m are terms ($m \geq 1$), then $A(t_1, \dots, t_m)$ is a term.

A *macro grammar* $G = (F, \Sigma, X, S, P)$ consists of a ranked alphabet F of nonterminals, a terminal alphabet Σ , a finite set $X = \{x_1, \dots, x_m\}$ of variables, where m is at least the rank of each symbol in F (variables and terminals have rank 0; F , Σ , and X are mutually disjoint), an initial nonterminal $S \in F_0$, and a finite set of productions or rules of the form $A(x_1, \dots, x_n) \rightarrow t$, where $A \in F_n$ and t is a term over $F \cup \Sigma \cup \{x_1, \dots, x_n\}$. (If $A \in F_0$, then the rule is $A \rightarrow t$ and t is a term over $F \cup \Sigma$).

We will always use a macro-grammar in the outside-in (OI) mode of derivation, i.e., the above rule can be applied only if A is not nested in another nonterminal. Application of the rule consists of replacing a subterm of the form $A(t_1, \dots, t_n)$, where t_i is a term, by $t[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n]$. Rules are applicable to terms over $F \cup \Sigma$, but, if needed, also to terms over $F \cup \Sigma \cup X$. For details, see [18, 14]. The language generated by G is $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$ as usual. The class of languages generated by all macro grammars is denoted OI.

In the rest of this section we mention some concepts and results from abstract automata theory, also called AFA theory, see [20].

Since the definitions in AFA theory are unduly complicated (thus obscuring the attractiveness of the field) we will instead use a notion of abstract automaton (or machine) suggested in [35, 19]. Since we will consider only the notion of a one-way acceptor, a class of such machines is completely determined by the structure of its storage (such as: pushdown, two counters, etc.), i.e., the set of storage configurations and the set of instructions manipulating the storage. Around 1966 three different definitions were proposed to catch this idea: the notion of machine of

[35], the notion of balloon automaton of [29], and the notion of AFA of [21]. Since the authors of AFA theory contributed the most to the field, the notion of AFA won the contest. It was shown in [19] that a variation of the machine notion of [35] would as well be used to establish the main results of AFA theory. Recently, in the wish to simplify AFA theory we “rediscovered” the machine notion of [35, 19], and, independently, the same happened in [23, 24, 25]. For a plea to use this notion rather than AFA we refer to the latter papers.

We start with the definition of storage and call it a machine type (other names could be: storage type, data structure, data store, automaton type, AFA-schema), cf. also Chapter X of [11].

1.1. DEFINITION. A *machine type* is a construct $D = (S, s_0, S_\infty, I, m)$, where

- S is a set of (storage) *configurations*,
- $s_0 \in S$ is the *initial configuration*,
- $S_\infty \subseteq S$ is the set of *final configurations*,
- I is a set of *instructions*,

— m is a mapping from I into the set of partial functions from S to S ; for $i \in I$, $m(i): S \rightarrow S$ is the meaning of i ; m is extended to I^* by interpreting concatenation as function composition (for $u, v \in I^*$, $m(uv) = \{(s_1, s_3) | (s_1, s_2) \in m(u) \text{ and } (s_2, s_3) \in m(v) \text{ for some } s_2 \in S\}$, $m(\lambda)$ is the identity function on S).

Furthermore we require that there exists $u \in I^*$ such that $(s_0, s) \in m(u)$ for some $s \in S_\infty$. ■

Tests on storage will be simulated (as usual) by having two instructions t and \bar{t} such that $m(t)$ is the identity on S_1 (where S_1 is the set of storage configurations for which the test is true) and undefined on $S - S_1$, and $m(\bar{t})$ is the identity on $S - S_1$ and undefined on S_1 . Then, “if test then A else B ” is simulated by “ $(t; A)$ or $(\bar{t}; B)$.”

The next definition is analogous to the one in Chapter 5 of [20].

1.2. DEFINITION. A machine type $D = (S, s_0, S_\infty, I, m)$ is *finitely encoded* if I is finite.

We now define the notion of a D -machine or machine of type D .

1.3. DEFINITION. Let $D = (S, s_0, S_\infty, I, m)$ be a machine type. A (one-way non-deterministic) *machine of type D* is a construct $M = (Q, \Sigma, I_M, q_0, Q_\infty, \delta)$, where Q is a finite set of states, Σ is the input alphabet, I_M is a finite subset of I , $q_0 \in Q$ is the initial state, $Q_\infty \subseteq Q$ is the set of final states, and the finite control δ is a finite subset of $Q \times \Sigma^* \times Q \times I_M^*$.

Acceptance by a machine M is defined in the usual way. Let $Q \times \Sigma^* \times S$ be the set of “total configurations” of M . The binary relation \vdash on the set of total con-

figurations is defined as follows: if $(q, w, p, u) \in \delta$ and $(s, t) \in m(u)$, then $(q, wy, s) \vdash (p, y, t)$ for every $y \in \Sigma^*$. The language accepted by M is $L(M) = \{w \in \Sigma^* \mid (q_0, w, s_0) \vdash^* (q, \lambda, s) \text{ for some } q \in Q_\infty \text{ and } s \in S\}$, where \vdash^* is the reflexive-transitive closure of \vdash . The class of languages defined by D is $K(D) = \{L(M) \mid M \text{ is a machine of type } D\}$. Two machine types D_1 and D_2 are equivalent if $K(D_1) = K(D_2)$. A machine type D is characterized by the following subset L_D of I^* .

1.4. DEFINITION. Let $D = (S, s_0, S_\infty, I, m)$ be a machine type. The set of successful instruction sequences of D is defined by $L_D = \{u \in I^* \mid (s_0, s) \in m(u) \text{ for some } s \in S_\infty\}$.

Since I is not necessarily finite, L_D is not necessarily a language. Note that the requirement in the last sentence of Definition 1.1 says that L_D is nonempty.

We need the following related results from AFA-theory. The proofs can easily be given by the reader, in particular when he is familiar with [20, 35, 19, 23, 24, 25].

1.5. THEOREM. For every machine type D , $K(D)$ is a full semi-AFL; in fact, $K(D)$ is the smallest full semi-AFL containing all $L_D \cap I_1^*$, where I_1 is a finite subset of I . If D is finitely encoded, then $K(D)$ is a full principal semi-AFL with generator L_D .

1.6. THEOREM. Let K be a class of languages:

- (i) K is a full semi-AFL iff there exists a machine type D such that $K(D) = K$.
- (ii) K is a full principal semi-AFL iff there exists a finitely encoded machine type D such that $K(D) = K$.

Finally, we need the following notion of nontriviality of a machine type (cf. p. 93 of [20]).

1.7. DEFINITION. A machine type $D = (S, s_0, S_\infty, I, m)$ is nontrivial if

- (1) it has a test on s_0 , i.e., there exist t_0 and \bar{t}_0 in I such that $m(t_0) = \{(s_0, s_0)\}$ and $m(\bar{t}_0) = \{(s, s) \mid s \in S - \{s_0\}\}$, and
- (2) there exists $s_1 \in S$ such that $s_1 \neq s_0$, $(s_0, s_1) \in m(u)$ for some $u \in I^*$, and $(s_1, s_\infty) \in m(v)$ for some $v \in I^*$ with $s_\infty \in S_\infty$.

The first condition in the above definition is one which most machine types satisfy (or, at least, addition of a test on s_0 would not harm them). The second condition is the real nontriviality condition. In general we may assume for any D that all its configurations s are "reachable" (i.e., $(s_0, s) \in m(u)$ for some $u \in I^*$), and "useful" (i.e., $(s, s_\infty) \in m(v)$ for some $v \in I^*$ and $s_\infty \in S_\infty$). In fact, we can drop from S all configurations which are not reachable or not useful without changing L_D and hence (Theorem 1.5) without changing $K(D)$. Now, condition (2) above says that there is at least one reachable and useful configuration apart from s_0 (which is, by Definition 1.1, always reachable and useful). Thus, if a machine type does not satisfy

this condition, then we may assume that $S = S_\infty = \{s_0\}$. This implies that L_D is I_1^* for some finite subset I_1 of I , and hence, by Theorem 1.5 again, $K(D) = \text{REG}$. Thus, any machine type (with a test on s_0) such that $K(D) \neq \text{REG}$ is nontrivial. For REG, both trivial and nontrivial machine types D (with a test on s_0) exist.

The main use of nontriviality is to store one bit of information in the data structure D . To "store s_0 or s_1 " we use the instruction sequences λ and u , respectively (assuming that D is in configuration s_0 initially, and that $(s_0, s_1) \in m(u)$). Then, later, to "test whether D is in s_0 or in s_1 ," we use the test on s_0 , and then, to bring D into a final configuration, we use the instruction sequences corresponding to s_0 or s_1 , respectively, which lead them into S_∞ (both are useful configurations).

We observe that in Theorem 1.6 we may always assume that the machine type is nontrivial.

We finally note that many variations of Theorem 1.6 can be shown (see [20]). If, e.g., I contains a reset instruction i_r such that $m(i_r) = \{(s, s_0) | s \in S_\infty\}$, then a machine characterization of full AFLs is obtained. If we restrict δ in every M to be a finite subset of $Q \times \Sigma \times Q \times I_M^*$, then a machine characterization of the (not necessarily full) semi-AFLs is obtained (such machines are called quasi-realtime [20]). We could also, dually, restrict δ to finite subsets of $Q \times \Sigma^* \times Q \times I_M$. Then we would characterize all prequasoids closed under union. Finally, if we assume that D has the structure of a pushdown of D_0 -tapes (where D_0 is another machine type), then we have a machine characterization of the (full) super-AFLs [26]. In Section 4 we will establish a similar result for the full hyper-AFLs, and for the classes $B(K)$.

2. EXTENDED BASIC MACRO GRAMMARS

In this section we define K -extended basic macro grammars for any class of languages K , and mention some of their elementary properties.

A macro grammar $G = (F, \Sigma, X, S, P)$ is basic if there are no nested nonterminals in the right-hand side of productions [18]. This means that each production is of the form

$$A(x_1, \dots, x_n) \rightarrow w_1 B_1(u_1) w_2 \cdots w_k B_k(u_k) w_{k+1}, \quad (*)$$

where A, B_1, \dots, B_k are nonterminals, w_i is a string over $\Sigma \cup \{x_1, \dots, x_n\}$, and u_i is a sequence of s_i strings over $\Sigma \cup \{x_1, \dots, x_n\}$, where s_i is the rank of B_i . G is linear basic if $k = 1$ or $k = 0$ in each such production [18].

To characterize L -systems by macro grammars, (finitely) extended linear basic macro grammars were introduced in [9]. This idea was taken over in [15], where the (finitely) extended basic macro grammars were defined: (generalized) macro grammars with productions of the form (*) in which every string w_i (and also those in u_i) is replaced by a finite language over $\Sigma \cup \{x_1, \dots, x_n\}$. In [7] the general case (for linear basic macro grammars) was studied, where each element of u_i and each w_i is replaced by a language from an arbitrary class K . We now do the same for the

basic macro grammars. As in [7], instead of putting L_i for w_i in (*) with $L_i \in K$ (and similarly for every element of u_i), we use an approach which allows us to view K -extended basic macro grammars as ordinary macro grammars (with infinitely many rules). Thus, we replace each w_i by a "language name" $\psi_i(x_1, \dots, x_n)$ and extend the set of productions by all rules $\psi_i(x_1, \dots, x_n) \rightarrow w$, with $w \in L_i$ (and similarly for each element of u_i). This leads to the following formal definitions.

2.1. DEFINITION. Let K be a class of languages. A K -extended basic macro grammar is a construct $G = (F, \Psi, \Sigma, X, S, d, P)$, where

F is a ranked alphabet of nonterminals;

Ψ is a ranked alphabet of language names;

Σ is the terminal alphabet;

$X = \{x_1, \dots, x_m\}$ is a finite set of variables; m is at least the rank of each symbol in $F \cup \Psi$ (terminals and variables have rank 0; the sets F, Ψ, Σ , and X are mutually disjoint);

$S \in F_0$ is the initial nonterminal;

d is a mapping $\Psi \rightarrow K$ such that, for $\psi \in \Psi_n$, $d(\psi) \subseteq (\Sigma \cup \{x_1, \dots, x_n\})^*$; $d(\psi)$ is the domain of ψ ;

P is a finite set of productions or rules each of the form

$A(\mathbf{x}) \rightarrow \psi_1(\mathbf{x}) B_1(\phi_1(\mathbf{x})) \psi_2(\mathbf{x}) B_2(\phi_2(\mathbf{x})) \cdots B_k(\phi_k(\mathbf{x})) \psi_{k+1}(\mathbf{x})$, where $(\mathbf{x}) = (x_1, \dots, x_n)$, $n \geq 0$, $A \in F_n$, $k \geq 0$, $B_i \in F$, $\psi_i \in \Psi_n$, and $(\phi_i(\mathbf{x})) = (\psi_{i1}(\mathbf{x}), \dots, \psi_{is}(\mathbf{x}))$ with $\psi_{ij} \in \Psi_n$ and s is the rank of B_i (thus s depends on i).

G is a K -extended linear basic macro grammar if $k = 0$ or $k = 1$ in all productions.

Whenever $d(\psi)$ is a singleton $\{w\}$, we use w rather than $\psi(\mathbf{x})$. Thus, as an example, $A(x_1, x_2) \rightarrow ax_1 B(x_2 x_2, \psi(x_1, x_2))$ abbreviates the production $A(x_1, x_2) \rightarrow \psi_1(x_1, x_2) B(\psi_2(x_1, x_2), \psi(x_1, x_2)) \psi_3(x_1, x_2)$ with $d(\psi_1) = \{ax_1\} = \{ax_1\}$, $d(\psi_2) = \{x_2 x_2\}$, and $d(\psi_3) = \{\lambda\}$. In fact, whenever we have a representation for a language $L \subseteq (\Sigma \cup \{x_1, \dots, x_n\})^*$, we may use that representation rather than a language name $\psi(x_1, \dots, x_n)$ with $d(\psi) = L$.

With each K -extended basic macro grammar $G = (F, \Psi, \Sigma, X, S, d, P)$ we associate an ordinary (outside-in) macro grammar G' with a countable rather than a finite number of rules, by viewing the language names as nonterminals, as follows: $G' = (F \cup \Psi, \Sigma, X, S, P')$, where $P' = P \cup \{\psi(x_1, \dots, x_n) \rightarrow w \mid n \geq 0, \psi \in \Psi_n, w \in d(\psi)\}$. By definition, the derivations of G are those of G' . The language generated by the K -extended basic macro grammar G is defined by $L(G) = L(G')$, where G' is the macro grammar associated with G as above. The class of all languages generated by K -extended {linear} basic macro grammars is denoted $B(K)$ ($LB(K)$, respectively).

As suggested in the Introduction, an extended basic macro grammar $G = (F, \Psi, \Sigma, X, S, d, P)$ may be viewed as an operation on languages. To be more precise, G without d is an operation: $L(G)$ is the result of applying this operation to the languages $d(\psi)$, $\psi \in \Psi$. Since these operations may be viewed as a generalization

of iterated substitutions, we will call them “basic substitutions.” Thus each extended basic macro grammar G (without d) is a *basic substitution*, and $L(G)$ is the result of applying this basic substitution G to the languages $d(\psi)$.

A *full basic-AFL* is a full AFL K such that $B(K) \subseteq K$. In other words, a full basic-AFL is a full AFL closed under basic substitutions. The analogous concept of a full AFL K such that $LB(K) \subseteq K$ coincides with that of a full hyper-AFL. This follows from the first part of the next proposition [7] (actually it holds under much weaker conditions on K). The second part of the proposition says that H has to be applied once only to obtain a full hyper-AFL [4].

2.2. PROPOSITION. *Let K be a prequasoid:*

- (i) $LB(K) = H(K)$,
- (ii) $H(K)$ is the smallest full hyper-AFL containing K .

Thus every full basic-AFL is a full hyper-AFL (in particular, it is substitution-closed).

To see why extended basic macro grammars really are basic macro grammars with languages from K in the arguments of their nonterminals, we need the notion of a language term and its domain. A *language term* of a K -extended basic macro grammar $G = (F, \Psi, \Sigma, X, S, d, P)$ is a term over Ψ . The *domain* $d(t)$ of a language term t is a language over Σ defined as follows: for $\psi \in \Psi_0$, $d(\psi)$ is given by d ; for $\psi \in \Psi_k$ and language terms t_1, \dots, t_k , $d(\psi(t_1, \dots, t_k)) = d(\psi)[x_1 \leftarrow d(t_1), \dots, x_k \leftarrow d(t_k)]$; finally, for language terms t_1 and t_2 , $d(t_1 t_2) = d(t_1) \cdot d(t_2)$. Note that $d(t)$ is an element of the closure of K under substitution and concatenation. Note also that $d(t) = \{w \in \Sigma^* \mid t \xrightarrow{*} w \text{ in } G\}$; a formal proof is left to the reader.

It is easy to see that every derivation of G (i.e., of the associated G') can be rearranged such that first only rules from P are applied and then rules from $\{\psi(x_1, \dots, x_n) \rightarrow w \mid w \in d(\psi)\}$. In the first stage a sentential form is a concatenation of language terms and of terms $B(t_1, \dots, t_n)$, where each t_i is a language term. Hence the first stage produces a language term t and in the second stage a string $w \in d(t)$ is produced. Formally, $L(G) = \bigcup \{d(t) \mid t \text{ is a language term, } S \xrightarrow{*} t \text{ in } G \text{ with rules from } P \text{ only}\}$. Hence during the first stage we can view the nonterminals as *holding languages in their arguments* (viz. $B(t_1, \dots, t_n)$ holds the languages $d(t_1), \dots, d(t_n)$), and also having languages in between them. Application of a rule $A(\mathbf{x}) \rightarrow \psi_1(\mathbf{x}) B_1(\phi_1(\mathbf{x})) \dots$, from P can be viewed as substitution of the actual (language) arguments of A into the formal (language) arguments of the B 's to obtain the actual (language) arguments of the B 's (and something similar for the ψ between the B 's). Then the first stage produces a language $d(t)$ and the second stage just picks a string from this language.

It should be clear from the above description of a derivation that it can be viewed as a generalized kind of iterated substitution (called “basic substitution” above). A formal treatment of these ideas is left to the reader; for the linear case see [7]. In fact, they are all based on the well-known fact that, in macro grammars, nesting of

terms corresponds to substitution of the languages generated by these terms (see the “parallel derivation lemmas” of [18]) and hence the productions of a macro grammar can be viewed as fixed-point equations, where the nonterminals stand for languages (over terminals and variables) and nesting is interpreted as substitution (of languages for the variables), see [9, 31, 14].

On of the consequences of the above is that if K is closed under substitution, then we may allow nested language names in the productions (because, e.g., $\psi_0(\psi_1(\mathbf{x}), \dots, \psi_k(\mathbf{x}))$ can be replaced by $\psi(\mathbf{x})$, where $d(\psi) = d(\psi_0)[x_1 \leftarrow d(\psi_1), \dots, x_k \leftarrow d(\psi_k)]$).

We now mention some easy consequences of the definitions. For arbitrary classes K_1 and K_2 , if $K_1 \subseteq K_2$, then $LB(K_1) \subseteq LB(K_2)$ and $B(K_1) \subseteq B(K_2)$. For arbitrary K , $K \subseteq LB(K) \subseteq B(K)$; to see that $K \subseteq LB(K)$, let $L \in K$ and consider the grammar with production $S \rightarrow \psi$ and $d(\psi) = L$; also $K \leftarrow K \subseteq LB(K)$: the grammar with productions $S \rightarrow A(\psi_1, \dots, \psi_n)$ and $A(\mathbf{x}) \rightarrow \psi_0(\mathbf{x})$ generates the language $d(\psi_0)[x_1 \leftarrow d(\psi_1), \dots, x_n \leftarrow d(\psi_n)]$. If ONE denotes the class of all singletons, then $B(\text{ONE})$ is the class of all basic macro languages and $LB(\text{ONE})$ is the class of all linear basic macro languages [18], i.e., we are back at the unextended case. Finally, $B(\text{FIN}) = \text{EB}$: the class of (finitely) extended basic macro grammars studied in [15], and $LB(\text{FIN}) = \text{ELB} = \text{ETOL} = H(\text{FIN})$: the class of (finitely) extended linear basic macro grammars [9]. Note that, formally, EB-grammars use only the language names $\emptyset \in \mathcal{P}_0$ (with $d(\emptyset) = \emptyset$) and $+ \in \mathcal{P}_2$ (with $d(+) = \{x_1, x_2\}$), but these may occur nested. Since FIN is closed under substitution, EB is clearly contained in $B(\text{FIN})$, see the remark above; on the other hand $B(\text{FIN}) \subseteq \text{EB}$ because by nesting $+$ and \emptyset any finite set can be obtained.

We end this section with two useful lemmas. The first concerns disposing of the language names outside and in-between nonterminals. A class K is closed under marking if aL and La are in K , whenever $L \in K$ and a does not occur in the alphabet of L .

2.3. LEMMA. *Let K be a class of languages closed under marking and with $\{\lambda\} \in K$. Every K -extended {linear} basic macro grammar is equivalent to one in which each production is of one of the forms (cf. Definition 2.1),*

$$A(\mathbf{x}) \rightarrow B_1(\phi_1(\mathbf{x})) B_2(\phi_2(\mathbf{x})) \cdots B_k(\phi_k(\mathbf{x}))$$

or

$$A(\mathbf{x}) \rightarrow \psi(\mathbf{x}).$$

Proof. We use the well-known trick [9] of putting the context into two extra arguments x_0 and x_∞ for each nonterminal. Every production $A(\mathbf{x}) \rightarrow \psi_1(\mathbf{x}) B_1(\phi_1(\mathbf{x})) \cdots B_k(\phi_k(\mathbf{x})) \psi_{k+1}(\mathbf{x})$ with $k \geq 1$ is replaced by the production $A(x_0, \mathbf{x}, x_\infty) \rightarrow B_1(x_0 \psi_1(\mathbf{x}), \phi_1(\mathbf{x}), \lambda) B_2(\psi_2(\mathbf{x}), \phi_2(\mathbf{x}), \lambda) B_3(\psi_3(\mathbf{x}), \phi_3(\mathbf{x}), \lambda) \cdots B_k(\psi_k(\mathbf{x}), \phi_k(\mathbf{x}), \psi_{k+1}(\mathbf{x}) x_\infty)$, where formally $x_0 \psi_1(\mathbf{x})$ should be replaced by $\psi'_1(\mathbf{x})$ with $d(\psi'_1) = x_0 d(\psi_1)$, and similarly for $\psi_{k+1}(\mathbf{x}) x_\infty$ (this is possible because K is

closed under marking). Note that formally λ (which also occurs between the B 's!) should also be replaced by a language name with domain $\{\lambda\}$. Every production $A(\mathbf{x}) \rightarrow \psi(\mathbf{x})$, i.e., $k=0$, is replaced by $A(x_0, \mathbf{x}, x_\infty) \rightarrow x_0\psi(\mathbf{x})x_\infty$. Finally, a new initial nonterminal S' is taken with production $S' \rightarrow S(\lambda, \lambda)$, where S is the old initial nonterminal (which now has two arguments x_0 and x_∞). Note that the construction preserves linearity. ■

The second technical lemma is needed for the case that K -extended basic macro grammars are considered, where K itself is also generated by (extended basic) macro grammars. The lemma shows that we can "turn the terminals of a macro grammar into variables." Thus, if G generates a language L over the terminal alphabet $\Sigma \cup \{x_1, \dots, x_n\}$, then we can change G such that Σ is its terminal alphabet, x_1, \dots, x_n are (additional) variables, and $S(x_1, \dots, x_n)$ generates L for some nonterminal S of rank n .

2.4. LEMMA. (i) *Let K be a class of languages which contains $\{x\}$ for every symbol x . Let $G = (F, \Psi, \Sigma, Y, S, d, P)$ be a K -extended $\{\text{linear}\}$ basic macro grammar and let $\{x_1, \dots, x_n\} \subseteq \Sigma$. Then there is a K -extended $\{\text{linear}\}$ basic macro grammar $G' = (F', \Psi', \Sigma - \{x_1, \dots, x_n\}, Y \cup \{x_1, \dots, x_n\}, S', d', P')$ such that $S \in F'_n$ and $\{w \in \Sigma^* \mid S(x_1, \dots, x_n) \xrightarrow{*} w \text{ in } G'\} = L(G)$.*

(ii) *The analogous fact holds for ordinary macro grammars.*

Proof. (i) Let $Y = \{y_1, \dots, y_m\}$ be the set of variables of G . To obtain G' we add the new arguments x_1, \dots, x_n to every nonterminal and language name of G ; thus $F' = F \cup \{S'\}$ (the nature of S' is immaterial) and $\Psi' = \Psi$ except that n is added to the ranks. These new arguments are just passed from nonterminal to nonterminal or language name and they are used in the domains of the language names. Thus the production $A(\mathbf{y}) \rightarrow \psi_1(\mathbf{y})B_1(\phi_1(\mathbf{y}))\dots$, in P is changed into the production $A(\mathbf{y}, \mathbf{x}) \rightarrow \psi_1(\mathbf{y}, \mathbf{x})B_1(\phi_1(\mathbf{y}, \mathbf{x}), \mathbf{x})\dots$, of P' (to have \mathbf{x} in the right-hand side, K should contain every $\{x_i\}$). The domains of the language names are unchanged ($d' = d$). It is left to the reader to prove formally that, in G' , $S(\mathbf{x})$ generates $L(G)$.

(ii) The proof is analogous; x_1, \dots, x_n are added as extra arguments and passed from nonterminal to nonterminal. ■

As an example of the use of the last lemma we show the following corollary.

2.5. COROLLARY. *OI is a full basic-AFL.*

Proof. Since it is known that OI is a full AFL, it remains to show that $B(\text{OI}) \subseteq \text{OI}$ (cf. the end of Sect. 3 of [7], where a similar proof is given of $LB(\text{OI}) \subseteq \text{OI}$). Let $G = (F, \Psi, \Sigma, X, S, d, P)$ be an OI-extended basic macro grammar, i.e., $d(\psi) \in \text{OI}$ for every $\psi \in \Psi$. Let G_ψ be a macro grammar with $L(G_\psi) = d(\psi)$. Now change G_ψ into a macro grammar G'_ψ according to Lemma 2.4(ii) such that, in G'_ψ , $\psi(x_1, \dots, x_n)$ generates the language $d(\psi)$, where $\psi \in \Psi_n$, i.e., ψ is also a nonterminal of G'_ψ of rank n . Finally, construct a new macro grammar G' by taking G and

all G'_ψ together. Then clearly $L(G') = L(G)$; whereas, in G , $\psi(x_1, \dots, x_n)$ produces a string $w \in d(\psi)$ in one step, now, in G' , $\psi(x_1, \dots, x_n)$ generates w using the productions of G'_ψ . ■

3. CLOSURE PROPERTIES

In this section we investigate the closure properties of $B(K)$ and $LB(K)$, for every prequasoid K . Apart from the fact that $B(K)$ and $LB(K)$ are full AFLs, we concentrated on their closure under several kinds of (iterated) substitution. In the next (main) theorem we show that $LB(K)$ is closed under iterated substitution (as we already know from Proposition 2.2), and that application of basic substitutions to $LB(K)$ does not lead out of $B(K)$. In general, $B(K)$ is not closed under basic substitution, not even under substitution (Theorem 3.7).

3.1. THEOREM. *For any prequasoid K , $B(LB(K)) = B(K)$ and $LB(LB(K)) = LB(K)$.*

Proof. The second equality follows immediately from Proposition 2.2. However, our proof of the first equality will show the second as a special case. It follows from $K \subseteq LB(K)$ that $B(K) \subseteq B(LB(K))$, and $LB(K) \subseteq LB(LB(K))$. We now show the reverse inclusions. Let $G = (F, \Psi, \Sigma, X, S, d, P)$ be an $LB(K)$ -extended basic macro grammar; we will show that there is an equivalent K -extended basic macro grammar G' (and if G is linear, so is G'). Let, for every $\psi \in \Psi$, $G(\psi)$ be a K -extended linear basic macro grammar generating $d(\psi)$, and let $G'(\psi)$ be obtained by turning the appropriate terminals x_1, \dots, x_n ($\psi \in \Psi_n$) into variables according to Lemma 2.4(i).

The main idea is as follows (cf. [9]). As argued in Section 2, $L(G) = \bigcup \{d(t) \mid t \text{ is a language term, } S \xrightarrow{*} t \text{ in } G \text{ with rules from } P\}$. Since each language name ψ in a language term t represents a language from $LB(K)$, $d(\psi)$ is itself of the form $\bigcup \{d(t') \mid \dots\}$, where the union is infinite in general. However, during derivation (in G) of a string from t , only finitely many strings from any $d(\psi)$ are actually used and hence we can replace $d(\psi)$ by a *finite union* of languages $d(t')$, for every derivation of G (formally this is based on the Δ -continuity of substitution, cf. [14]). Thus our new grammar G' will simulate a derivation of G , but instead of using a language name $\psi(x_1, \dots, x_n)$, it will first simulate finitely many derivations of the linear grammar $G'(\psi)$, accumulating the so-derived language(term)s, and then continue simulating the derivation of G .

The formal construction is as follows (see Theorem 1.4.2 in Chap. 4 of [9], where it is shown by this technique that ETOL is closed under iterated substitution; see also Lemma 2.2 of [15], where it is shown that $B(\text{REG}) = B(\text{FIN})$): We assume that G and all $G'(\psi)$ are in the normal form of Lemma 2.3 (clearly, since K is closed under marking, so is $LB(K)$). Consider first a production $p: A(\mathbf{x}) \rightarrow B_1(\phi_1(\mathbf{x})) \cdots B_k(\phi_k(\mathbf{x}))$ of G . In G' we replace this production by a set of productions which can

generate any finite approximation to the $\phi_j(\mathbf{x})$ in some new arguments. Thus we first simulate any finite number of derivations for each of the ψ occurring in ϕ_j , before passing the results to B_j . Consider some $B(\psi_1(\mathbf{x}), \dots, \psi_n(\mathbf{x}))$ in the right-hand side of the above rule (where $B = B_j$ and $(\psi_1(\mathbf{x}), \dots, \psi_n(\mathbf{x})) = (\phi_j(\mathbf{x}))$). Let S_i be the nonterminal of $G'(\psi_i)$ such that $S_i(\mathbf{x})$ generates $d(\psi_i)$. G' contains all nonterminals of G , and new nonterminals E_i^C with arguments (\mathbf{z}, \mathbf{y}) , where $1 \leq i \leq n$, C is any nonterminal of $G'(\psi_i)$, \mathbf{z} is the sequence of arguments of C (note that \mathbf{x} is a subsequence of \mathbf{z} by the construction of Lemma 2.4), and $\mathbf{y} = y_1, \dots, y_n$ are new variables (which will contain approximations of $d(\psi_1), \dots, d(\psi_n)$, respectively). Note that for every production p and for every j ($1 \leq j \leq k$), G' contains a different collection of nonterminals E_i^C , i.e., E_i^C depends additionally on p and j . The language names of G' are those of all $G'(\psi)$, with n added to their rank and with the same domains. In G' we keep the above rule in which, however, every $B(\psi_1(\mathbf{x}), \dots, \psi_n(\mathbf{x}))$ is replaced by the corresponding $E_1^{S_1}(\mathbf{x}, \emptyset)$, where \emptyset is a sequence of n (new) language names, each with empty domain. For the new nonterminals E_i^C we add all rules $E_i^C(\mathbf{z}, y_1, \dots, y_n) \rightarrow \text{r.h.s.}$ to G' , where

(1) if $C(\mathbf{z}) \rightarrow D(\phi(\mathbf{z}))$ is a production in $G'(\psi_i)$, then $\text{r.h.s.} = E_i^D(\phi(\mathbf{z}, \mathbf{y}), y_1, \dots, y_n)$;

(2) if $C(\mathbf{z}) \rightarrow \psi(\mathbf{z})$ is a production in $G'(\psi_i)$, then $\text{r.h.s.} = E_i^{S_i}(\mathbf{x}, y_1, \dots, y_i \cup \psi(\mathbf{z}, \mathbf{y}), \dots, y_n)$, where $y_i \cup \psi(\mathbf{z}, \mathbf{y})$ should formally be replaced by a new $\psi'(\mathbf{z}, \mathbf{y})$ with domain $\{y_i\} \cup L$, where L is the domain of ψ ; note that \mathbf{x} is a subsequence of \mathbf{z} ;

(3) if $C(\mathbf{z}) \rightarrow \psi(\mathbf{z})$ is a production in $G'(\psi_i)$ and $i < n$, then $\text{r.h.s.} = E_{i+1}^{S_{i+1}}(\mathbf{x}, y_1, \dots, y_i \cup \psi(\mathbf{z}, \mathbf{y}), \dots, y_n)$;

(4) if $C(\mathbf{z}) \rightarrow \psi(\mathbf{z})$ is a production in $G'(\psi_i)$ and $i = n$, then $\text{r.h.s.} = B(y_1, \dots, y_{n-1}, y_n \cup \psi(\mathbf{z}, \mathbf{y}))$.

Thus, the approximations are built for ψ_1, \dots, ψ_n in that order and then they are passed to B .

Finally, if $A(\mathbf{x}) \rightarrow \psi(\mathbf{x})$ is a rule in G , then we add the rule $A(\mathbf{x}) \rightarrow S_\psi(\mathbf{x})$ and all productions of $G'(\psi)$ to G' , where S_ψ is the nonterminal of $G'(\psi)$ such that $S_\psi(\mathbf{x})$ generates $d(\psi)$. This ends the construction of G' . It should be clear that G' is a K -extended basic macro grammar and is linear if G is. A formal proof of $L(G') = L(G)$ is left to the reader. ■

3.2. COROLLARY. For every prequasoid K , $B(K)$ is closed under $LB(K)$ -substitution.

Proof. We have to show that $B(K) \leftarrow LB(K) \subseteq B(K)$. Let G be a K -extended basic macro grammar and turn all its terminals a_1, \dots, a_n into variables according to Lemma 2.4. Let G' be the resulting grammar and S the nonterminal such that $S(a_1, \dots, a_n)$ generates $L(G)$. Let $L_i \in LB(K)$ for $1 \leq i \leq n$. Construct G_1 from G' by adding the rule $S_1 \rightarrow S(\psi_1, \dots, \psi_n)$, where ψ_i has domain L_i . Then $L(G_1) = L(G)[a_1 \leftarrow L_1, \dots, a_n \leftarrow L_n]$. G_1 is an $LB(K)$ -extended basic macro grammar. Hence, by Theorem 3.1, $L(G_1) \in B(K)$. ■

In fact, Theorem 3.1 implies that $B(K)$ is even closed under iterated $LB(K)$ -substitution. To show this we need a lemma relating iterated substitution to LB , cf. [7]. For the sake of this lemma, let $LB(K, K_0)$ denote the class of languages generated by $(K \cup K_0)$ -extended linear basic macro grammars in normal form (according to Lemma 2.3) such that all language names in the nonfinal rules (i.e., in the ϕ_j) have domains in K , whereas the language names in the final rules (i.e., the ψ) have domains in K_0 (cf. the statement of Lemma 2.3).

3.3. LEMMA. *Let K and K_0 be classes of languages closed under homomorphisms, and let K contain \emptyset and $\{a\}$ for every symbol a . Then $H(K_0, K) \subseteq LB(K, K_0)$.*

Proof. The proof goes by the usual simulation of iterated substitution by macro grammars [9, 7]. Let $G = (V, \Sigma, U, A)$ be a K -iteration grammar with axiom set $A \in K_0$. U is a set of K -substitutions on $V = \{a_1, \dots, a_n\}$. We construct an extended linear basic macro grammar $G' = (\{S, B\}, \Psi, \Sigma, \{x_1, \dots, x_n\}, S, d, P)$ with the following productions:

(1) $S \rightarrow B(\psi_1, \dots, \psi_n)$ with $d(\psi_i) = \{a_i\}$ if $a_i \in \Sigma$, and \emptyset if $a_i \notin \Sigma$;

(2) for every K -substitution $f \in U$, $B(x_1, \dots, x_n) \rightarrow B(\psi_1(\mathbf{x}), \dots, \psi_n(\mathbf{x}))$, where $d(\psi_i) = h(f(a_i))$ and h is the homomorphism that replaces a_1, \dots, a_n by x_1, \dots, x_n , respectively;

(3) $B(x_1, \dots, x_n) \rightarrow \psi(x_1, \dots, x_n)$ with $d(\psi) = h(A)$.

If $B(t_1, \dots, t_n)$ is a sentential form of G' , then there exists $u \in U^*$ such that $d(t_i) = u(a_i) \cap \Sigma^*$, and vice versa. Thus $H(K_0, K) \subseteq LB(K, K_0)$. ■

3.4. THEOREM. *For every prequasoid K , $B(K)$ is closed under iterated $LB(K)$ -substitution.*

Proof. We have to prove that $H(B(K), LB(K)) \subseteq B(K)$. Since (as can easily be seen) $B(K)$ and $LB(K)$ satisfy the assumptions of Lemma 3.3., $H(B(K), LB(K)) \subseteq LB(LB(K), B(K))$. Using Lemma 2.4 (to replace in final rules the language names with domain in $B(K)$ by their grammars), it follows that $LB(LB(K), B(K)) \subseteq B(LB(K) \cup K) \subseteq B(LB(K))$. The theorem now follows from Theorem 3.1. ■

As far as substitution of $B(K)$ languages into other languages is concerned, we have the following easy result:

3.5. THEOREM. *For every prequasoid K , $B(K)$ is closed under substitution into CF, i.e., $CF \leftarrow B(K) \subseteq B(K)$.*

Proof. Replace in the given context-free grammar every terminal a by the initial nonterminal of the K -extended basic macro grammar generating the language to be substituted for a . Note that all productions of the context-free grammar are allowable " K -extended productions," because K contains all singleton languages. ■

We now prove that $B(K)$ is a full AFL.

3.6. THEOREM. *For every prequasoid K , $B(K)$ is a full AFL.*

Proof. To show closure under concatenation, union, and Kleene star it suffices to prove closure under substitution into the regular languages. This follows from Theorem 3.5. It now suffices to show closure under regular substitution and intersection with regular languages. Closure under regular substitution follows from Corollary 3.2 (note that $REG \subseteq LB(FIN) \subseteq LB(K)$). Closure under intersection with regular languages can be proved in the same way as was done for $EB = B(FIN)$ in Lemma 4.2 of [15]. For the interested reader we note that the proof is, in fact, precisely the same (first use Lemma 2.3), except that, in the notation used there, S_i is now a language from K and $\{w \in \phi(S_i) \mid h(w) = g\} = \phi(S_i) \cap \{w \mid h(w) = g\}$ is again in K because ϕ is a finite substitution and $\{w \mid h(w) = g\}$ is regular. ■

From Proposition 2.2 we know that $LB(K)$ is a full hyper-AFL. In the next theorem we show that $B(K)$ need not be closed under substitution. In fact, the results on substitution of Corollary 3.2 and Theorem 3.5 are optimal for $B(K)$, in general. To show this we use the particular type of substitution introduced in [27] to show a similar result for the one-way stack languages (and after that used in the “syntactic lemma” of [28] to investigate the closure under substitution of AFLs).

For any language L , τ_L denotes the substitution defined by $\tau_L(a) = aL$ for every symbol a in the alphabet under consideration. Thus, for languages L_1 and L_2 , $\tau_{L_2}(L_1)$ denotes the language $L_1[a \leftarrow aL_2]$, where a ranges over the alphabet of L_1 . We only consider the case that the alphabets of L_1 and L_2 are disjoint.

3.7. THEOREM. *Let K be a prequasoid and let L_1 and L_2 be languages over disjoint alphabets Σ_1 and Σ_2 , respectively. If $\tau_{L_2}(L_1) \in B(K)$, then $L_1 \in CF$ or $L_2 \in LB(K)$.*

Proof. The proof follows the same idea as in Lemma 4.1 of [27]. Let $G = (F, \Psi, \Sigma, X, S, d, P)$ be a K -extended basic macro grammar such that $L(G) = \tau_{L_2}(L_1)$. We assume G to be in the normal form of Lemma 2.3. Furthermore we assume that G is \emptyset -free (i.e., $d(\psi)$ is nonempty for all ψ) and reduced (in the sense that the context-free grammar obtained from G by erasing all arguments, is reduced). The latter two properties ensure that every sentential form of G generates at least one string. It is left to the reader to do the (standard) constructions for obtaining these two properties.

Let G_{lin} be the K -extended linear basic macro grammar obtained by “breaking G into linear pieces,” i.e., $G_{lin} = (F, \Psi, \Sigma, X, S, d, P')$, where P' contains the following rules:

- (1) if $A(\mathbf{x}) \rightarrow B_1(\phi_1(\mathbf{x})) \cdots B_k(\phi_k(\mathbf{x}))$ is in P , then $A(\mathbf{x}) \rightarrow B_i(\phi_i(\mathbf{x}))$ is in P' for all i , $1 \leq i \leq k$;
- (2) if $A(\mathbf{x}) \rightarrow \psi(\mathbf{x})$ is in P , then it is also in P' .

We now consider two cases.

Case 1. For every string $w \in L_2$ there exist $a, b \in \Sigma_1$ and $u, v \in (\Sigma_1 \cup \Sigma_2)^*$ such that $uawbv \in L(G_{\text{lin}})$. Since G is \emptyset -free and reduced, G_{lin} produces only substrings of strings of $L(G)$. Hence, in this case, $L_2 = T(L(G_{\text{lin}}))$, where T is the *ngsm* which extracts from a given string all substrings over Σ_2 which are surrounded by two symbols from Σ_1 . Since $L(G_{\text{lin}}) \in LB(K)$ and $LB(K)$ is closed under *ngsm* mappings (Proposition 2.2), $L_2 \in LB(K)$.

Case 2. This is the negation of Case 1, i.e., there exists $w \in L_2$ such that awb is a not a substring of any string of $L(G_{\text{lin}})$, for any $a, b \in \Sigma_1$. We want to show that in this case $L_1 \in CF$. Let $a_1 a_2 \cdots a_n$ be any string in L_1 (with $a_i \in \Sigma_1$) and consider a derivation of $u = a_1 w a_2 w \cdots a_n w$ in G (where w is the special string from L_2). If, during this derivation, a nonterminal A contains in one of its arguments a language term t and $d(t)$ contains a string v , then either v does not occur as a substring of u (i.e., it is discarded) or v contains at most one symbol a_i (otherwise, "following" this string v in the derivation, we could construct a derivation of G_{lin} generating a string with at least two occurrences of symbols a_i , surrounding w , contradicting the assumption of this case). Hence, since we are now only interested in generating $a_1 a_2 \cdots a_n$, we can restrict the arguments to strings which contain at most one symbol from Σ_1 . Thus, to generate L_1 , we would like to define a new grammar G' such that if A contains the language L in one of its arguments (during derivation in G), then in G' it contains the (finite) language $h(L) \cap (\Sigma_1 \cup \{\lambda\})$, where h is the homomorphism which erases the elements of Σ_2 and is the identity on Σ_1 . By the above argument, such a G' would generate L_1 . In fact, since the involved languages have bounded cardinality, we can construct G' as a context-free grammar. Formally, G' has nonterminals of the form $A(r_1, \dots, r_n)$, where A is a nonterminal of G of rank n and $r_i \in \Sigma_1 \cup \{\lambda\}$ for $1 \leq i \leq n$. G' has initial nonterminal S and terminal alphabet Σ_1 . The productions of G' are as follows:

- (1) if $A(x_1, \dots, x_n) \rightarrow \dots B_i(\psi_1(\mathbf{x}), \dots, \psi_m(\mathbf{x})) \dots$ is in P , then $A(r_1, \dots, r_n) \rightarrow \dots B_i(s_1, \dots, s_m) \dots$ is a production of G' , where, for $1 \leq j \leq m$, $s_j = h(d(\psi_j))[x_1 \leftarrow r_1, \dots, x_n \leftarrow r_n] \cap (\Sigma_1 \cup \{\lambda\})$;
- (2) if $A(x_1, \dots, x_n) \rightarrow \psi(\mathbf{x})$ is in P , then $A(r_1, \dots, r_n) \rightarrow w$ is a production of G' for every w in $h(d(\psi))[x_1 \leftarrow r_1, \dots, x_n \leftarrow r_n] \cap (\Sigma_1 \cup \{\lambda\})$.

This ends the construction of G' . It is left to the reader to prove formally that $L(G') = L_1$, and hence $L_1 \in CF$. ■

3.8. COROLLARY. Let K_1, K_2 , and K be prequasoids, with $K_i \subseteq B(K)$ for $i = 1, 2$:

- (i) $K_1 \leftarrow K_2 \subseteq B(K)$ iff $K_1 \subseteq CF$ or $K_2 \subseteq LB(K)$;
- (ii) $B(K) \leftarrow K_2 \subseteq B(K)$ iff $K_2 \subseteq LB(K)$;
- (iii) if $LB(K) \not\subseteq B(K)$, then $K_1 \leftarrow B(K) \subseteq B(K)$ iff $K_1 \subseteq CF$;
- (iv) $LB(K)$ is the largest substitution-closed class of languages contained in $B(K)$.

Proof. (i) In the if direction the statement follows from Corollary 3.2 and Theorem 3.5. Let us show the only-if direction. Suppose that K_1 is not contained in CF and let $L_1 \in K_1 - CF$. Consider any $L_2 \in K_2$. We may assume that the alphabets of L_1 and L_2 are disjoint. Clearly, $aL_2 \in K_2$ for every symbol a . Hence $\tau_{L_2}(L_1) \in K_1 \leftarrow K_2$ and so $\tau_{L_2}(L_1) \in B(K)$. Then, since $L_1 \notin CF$, $L_2 \in LB(K)$ by Theorem 3.7. And so $K_2 \subseteq LB(K)$.

(ii) From (i) follows that $B(K) \leftarrow K_2 \subseteq B(K)$ if and only if $B(K) \subseteq CF$ or $K_2 \subseteq LB(K)$. But $B(\text{FIN})$ contains a noncontext-free language.

(iii) is immediate from (i).

(iv) If $K_1 \leftarrow K_1 \subseteq K_1 \subseteq B(K)$, then, by (i), $K_1 \subseteq CF$ or $K_1 \subseteq LB(K)$. But $CF \subseteq \text{ETOL} = LB(\text{FIN}) \subseteq LB(K)$, and so $K_1 \subseteq LB(K)$. ■

As a consequence, the results on substitution of Corollary 3.2 and Theorem 3.5 are optimal for $B(K)$ whenever $LB(K) \not\subseteq B(K)$. This is, in particular, true for $K = \text{FIN}$: in [15] it is shown that $\text{ETOL} = LB(\text{FIN}) \not\subseteq B(\text{FIN}) = \text{EB}$.

3.9. COROLLARY. *$EB = B(\text{FIN})$ is not closed under substitution. $\text{ETOL} = LB(\text{FIN})$ is the largest substitution-closed class of languages contained in EB .*

Very similar results on substitution were proved in [27] for the class Stack of stack languages and the class CStack of checking-stack languages (with Stack, CStack, and CF instead of $B(\text{FIN})$, $LB(\text{FIN})$ and CF , respectively). For Stack this implies that it has two maximal substitution-closed classes (viz. CF and CStack) whereas $B(\text{FIN})$ contains just one. Note that $B(\text{FIN})$ and $LB(\text{FIN})$ are recognized by the stack-pushdown machine and the checking-stack-pushdown machine, respectively (see [15] and Sect. 5).

Since $B(K)$ is, in general, not a full basic-AFL, the smallest full basic-AFL containing K has to be obtained by iterating the B -operation.

3.10. DEFINITION. For any class K of languages, $B^*(K) = \bigcup \{B^n(K) \mid n \geq 0\}$.

Clearly, if K is a prequasoid, then $B^n(K)$ is a nondecreasing sequence of full AFLs (Theorem 3.6), and $B^*(K)$ is the smallest full basic-AFL containing K . Hence $B^*(\text{FIN})$ is the smallest full basic-AFL. By Corollary 3.9 it properly contains $B(\text{FIN})$, and by Corollary 2.5 it is contained in OI. In the next theorem we give a characterization of $B^*(\text{FIN})$ in terms of macro grammars; it equals the class $\text{Ult } B$ of languages generated by ultra-basic macro grammars. These grammars were introduced in [16] as generalized basic macro grammars with a restriction on the way nonterminals may be nested (just as ultralinear context-free grammars generalize linear context-free grammars). It was shown in [16] that $\text{Ult } B$ is properly contained in OI.

3.11. DEFINITION. A macro grammar $G = (F, \Sigma, X, S, P)$ is *ultrabasic* if there exists a mapping

$$\text{level: } F \rightarrow \{1, 2, \dots, k\} \quad \text{for some } k \geq 1,$$

such that if $A(x_1, \dots, x_n) \rightarrow t$ is a rule in P , then

- (i) if a nonterminal B occurs in t , then $\text{level}(B) \leq \text{level}(A)$;
- (ii) if a nonterminal B occurs in the argument of another nonterminal in t , then $\text{level}(B) < \text{level}(A)$.

Ult B denotes the class of languages generated by ultrabasic macro grammars.

3.12. THEOREM. $\text{Ult } B = B^*(\text{FIN})$.

Proof. (i) $B^*(\text{FIN}) \subseteq \text{Ult } B$. Since clearly $\text{FIN} \subseteq \text{Ult } B$, it suffices to prove that $B(\text{Ult } B) \subseteq \text{Ult } B$. Consider an Ult B -extended basic macro grammar $G = (F, \Psi, \Sigma, X, S, d, P)$, i.e., $d(\psi) \in \text{Ult } B$ for all ψ . Let $G(\psi)$ be the ultrabasic macro grammar generating $d(\psi)$, and turn its terminals x_1, \dots, x_n into variables ($\psi \in \Psi_n$), cf. Lemma 2.4. Let $G'(\psi)$ be the resulting grammar and let ψ be the nonterminal of $G'(\psi)$ such that $\psi(x_1, \dots, x_n)$ generates $d(\psi)$ in $G'(\psi)$. Now we define a new macro grammar G' by taking together G and all $G'(\psi)$. Clearly $L(G') = L(G)$. Moreover, G' is also ultrabasic. Let k be the maximal level of nonterminals in the $G'(\psi)$. Define the level of the nonterminals of G to be $k + 1$. Clearly this new level function works for G' .

(ii) $\text{Ult } B \subseteq B^*(\text{FIN})$. We show that $L(G) \in B^*(\text{FIN})$ for every ultrabasic macro grammar $G = (F, \Sigma, X, S, P)$ by induction on the maximal level k . If $k = 1$, then G is an ordinary basic macro grammar and so $L(G) \in B(\text{FIN})$. Assume now that the result holds for all values less than k , and let G have maximal level k . We construct a $B^*(\text{FIN})$ -extended basic macro grammar G' as follows (since $B^*(\text{FIN})$ is closed under substitution, we allow nested language names in the productions of G' , see Sect. 2): The nonterminals of G' are the level k nonterminals of G . The language names of G' are all other nonterminals of G , and, for such ψ , $d(\psi) = \{w \in (\Sigma \cup \{x_1, \dots, x_n\})^* \mid \psi(x_1, \dots, x_n) \xrightarrow{*} w \text{ in } G\}$. The productions of G' are those of G which have a level k nonterminal in the left-hand side. It should be clear that $L(G') = L(G)$. By induction $d(\psi) \in B^*(\text{FIN})$, because in derivations starting with $\psi(x_1, \dots, x_n)$ only nonterminals with level $< k$ are used. Hence $L(G) = L(G') \in B(B^*(\text{FIN})) = B^*(\text{FIN})$. ■

3.13. COROLLARY. $\text{ETOL} = \text{LB}(\text{FIN}) \subsetneq \text{EB} = \text{B}(\text{FIN}) \subsetneq \text{Ult } B = B^*(\text{FIN}) \subsetneq \text{OI}$.

Proof. For $\text{ETOL} \subsetneq \text{EB}$, see [15], and for $\text{Ult } B \subsetneq \text{OI}$, see [16]. $B(\text{FIN}) \subsetneq B^*(\text{FIN})$ follows from Corollary 3.9. ■

In fact, since EB is not closed under substitution, there is an infinite hierarchy of full AFLs between EB and $B^*(\text{FIN})$, see [28]. In [13] it will be shown that there

is even an infinite hierarchy of full hyper-AFLs between EB and $B^*(\text{FIN})$. Note that in [10] it was first shown that $\text{ETOL} \subsetneq \text{OI}$, which solved an open problem of that time. We finally note that it was proved in [26] that OI is not the least super-AFL containing Stack, i.e., OI cannot be reached from Stack by nested iterated substitution (and full AFL operations). Since $\text{Stack} \subseteq \text{EB}$ (see [15]), Corollary 3.13 shows that OI cannot be reached from Stack even by iterated substitution or basic substitution.

4. STACK CONTROLLED MACHINES

In this section we generalize the s-pd machine of [15] to arbitrary machine types, and show that these characterize all $B(K)$, where K is a full semi-AFL. As a special case, the generalized cs-pd machines characterize the full hyper-AFLs. See Section 1 for terminology on machine types.

For every machine type D we will consider a new machine type: the stack-controlled machine type corresponding to D . A configuration of the stack controlled machine type (see Fig. 1) consists of an ordinary stack together with a pushdown of D -tapes (i.e., a pushdown each element of which is a configuration of D).

The pushdown is synchronized with the stack: it is upside down with respect to the stack; its bottom is one square below the top of the stack and its top follows the movements of the stack pointer, whenever the stack pointer reads in the stack. Thus each D -tape of the pushdown is associated to a stack square. The pushdown is empty if and only if the pointer is at the top of the stack. At each moment the machine has access to the stack symbol pointed at and to the lowest D -tape (i.e., the D -tape on the top of the pushdown). In one move the machine can change its

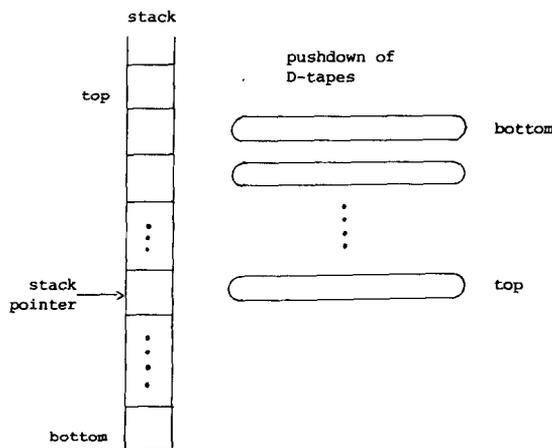


FIG. 1. A stack controlled pushdown of D -tapes.

configuration in one of the following ways: (1) with the stack pointer at the top of the stack it can push and pop as usual; (2) it can use any instruction of D to change the lowest D -configuration; (3) it can move down the stack pointer one square and simultaneously "open a new D -tape" by pushing the initial D -configuration on top of the pushdown; finally (4) it can move up the stack pointer one square and simultaneously "close the D -tape" by popping the lowest D -configuration off the pushdown, provided it is a final D -configuration. Thus, the pushdown of D -tapes can be used to simulate computations of machines of type D : they can be started (by opening a new D -tape), continued in a piecewise fashion (each time that D -tape is on top of the pushdown), and ended (by closing the D -tape). We note that the pushdown part of the machine is exactly the same as the nested AFA of [26]. In case each D -tape is just one square containing a symbol, the machine is the s-pd machine of [15].

For technical reasons it is convenient to add one extra facility to the stack controlled machine type, viz., to have one (additional) square associated with each D -tape that may contain any symbol (see Fig. 2). In other words, an ordinary pushdown store grows together with the pushdown of D -tapes. At each moment the machine has also access to the (topmost) pushdown symbol, and it should push and pop pushdown symbols when moving down and up, respectively. Thus, when disregarding the D -tapes, the machine is now precisely the s-pd machine of [15]. The main use of the extra pushdown square is to store finite information on a computation of a D -machine which is simulated (piecemeal) on the corresponding D -tape (such as the state of the machine). We will denote the machine type of Fig. 1 by $S(D)$ and that of Fig. 2 by $SP(D)$, because when disregarding the D -tapes, one is a stack machine and the other a stack-pushdown machine. Note however that both machine types are stack controlled pushdowns of D -tapes. We will see later that the two machine types are equivalent for nontrivial D .

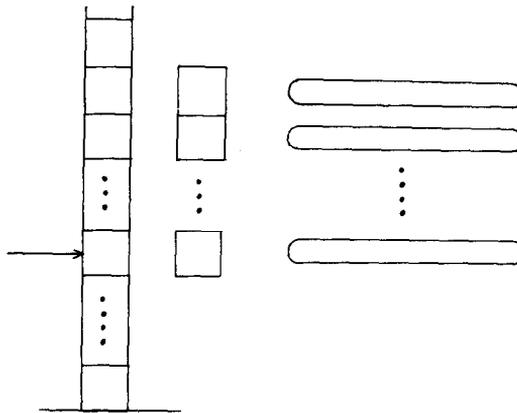


FIG. 2. A stack controlled pushdown of D -tapes with extra pushdown squares.

We now proceed with the formal definitions. Let Γ be a fixed (possibly infinite) set of symbols (from which the stack symbols and the pushdown symbols are taken).

4.1. DEFINITION. Let $D = (S, s_0, S_\infty, I, m)$ be a machine type. The *stack controlled machine type* corresponding to D , denoted by $SP(D)$, is $(S', s'_0, S'_\infty, I', m')$, where

$$S' = \Gamma^* \cup \Gamma^*(\Gamma \times \Gamma \times S)^+ \Gamma,$$

$$s'_0 = \lambda,$$

$$S'_\infty = \{\lambda\},$$

I' consists of

- (i) all elements of I ,
- (ii) all instructions $\text{push}(\gamma)$, pop , $\text{movedown}(\gamma)$, moveup , for $\gamma \in \Gamma$,
- (iii) all tests stackempty , pdempty , $\text{stacksymbol} = \gamma$, $\text{pdsymbol} = \gamma$, for $\gamma \in \Gamma$, (note that, as explained in Sect. 4, a test really consists of two instructions which are complementary partial identities).

We require the instructions of I to be disjoint with those in (ii) and (iii) (if necessary, this can be achieved by renaming). The meaning of the instructions is defined using the following notational conventions: $\gamma \in \Gamma$, $w \in \Gamma^*$, $s \in S$, $\alpha \in (\Gamma \times \Gamma \times S)^*$, possibly with subscripts and primes. Intuitively S contains two types of configurations: $w \in \Gamma^*$ denotes a stack with the pointer at the top (i.e., at the rightmost symbol of w); and $w \langle \gamma_1, \gamma'_1, s_1 \rangle \cdots \langle \gamma_n, \gamma'_n, s_n \rangle \gamma \in \Gamma^*(\Gamma \times \Gamma \times S)^+ \Gamma$ ($n \geq 1$) denotes the configuration shown in Fig. 3. We now define m' . For $i \in I$,

$m'(i) = \{(w \langle \gamma_1, \gamma'_1, s \rangle \alpha \gamma, w \langle \gamma_1, \gamma'_1, t \rangle \alpha \gamma) \mid (s, t) \in m(i)\}$, i.e., i operates on the lowest D -tape;

$$m'(\text{push}(\gamma)) = \{(w, w\gamma) \mid w \in \Gamma^*\};$$

$$m'(\text{pop}) = \{(w\gamma, w) \mid w \in \Gamma^*, \gamma \in \Gamma\}, \text{ as usual};$$

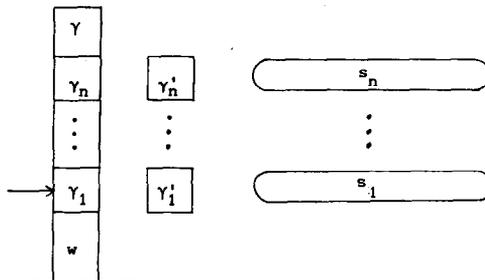


FIG. 3. The configuration $w \langle \gamma_1, \gamma'_1, s_1 \rangle \cdots \langle \gamma_n, \gamma'_n, s_n \rangle \gamma$ of $SP(D)$.

$m'(\text{movedown}(\gamma)) = \{(w\gamma_1\alpha\gamma_2, w\langle\gamma_1, \gamma, s_0\rangle\alpha\gamma_2)\}$, where s_0 is the initial configuration of D ;

$$m'(\text{moveup}) = \{(w\langle\gamma_1, \gamma'_1, s\rangle\alpha\gamma, w\gamma_1\alpha\gamma) \mid s \in S_\infty\}.$$

For a configuration $s' \in S'$, stackempty is true iff $s' = \lambda$, pdempty is true iff $s' \in \Gamma^*$; $\text{stacksymbol} = \gamma$ is true iff $s' = w\gamma_1$ or $s' = w\langle\gamma_1, \gamma'_1, s_1\rangle\alpha\gamma_2$ with $\gamma_1 = \gamma$; and $\text{pdsymbol} = \gamma$ is true iff $s' = w\langle\gamma_1, \gamma'_1, s_1\rangle\alpha\gamma_2$ and $\gamma'_1 = \gamma$. ■

As a first, easy, property we show that Γ may be taken finite.

4.2. LEMMA. *For any machine type D , $K(SP(D))$ does not depend on Γ , provided Γ has at least two elements.*

Proof. The usual trick of coding the stack and pushdown symbols as strings over the alphabet $\{0, 1\}$ can be used. Let k be the length of these strings. If, during some computation, a D -tape is associated with some stack square, then, in the simulating computation, this D -tape is associated with the topmost square of the k squares which code the original square, see Fig. 4. The D -tapes associated with the $k - 1$ other squares are dummy tapes (they are opened, brought into a final configuration by some $u \in L_D$, and closed). It should be clear that in this way the current stack symbol and pdsymbol can easily be decoded. The details are left to the reader. ■

Thus, if D is finitely encoded, we may assume that $SP(D)$ is also finitely encoded.

For stack-like machines an important restricted machine type is its checking-stack version. A checking-stack machine is a stack machine which first executes any number of pushes, then any number of movedowns and moveups, and finally any number of pops [27]; s-pd machines were actually defined as a generalization of

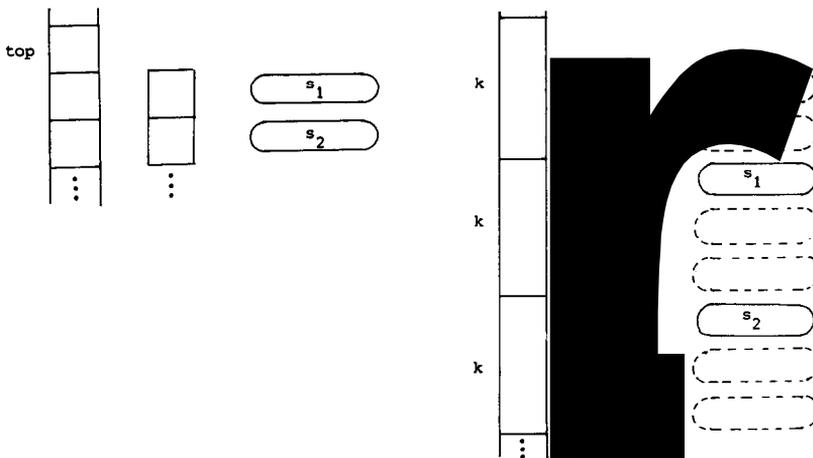


FIG. 4. Binary coding of Γ .

the cs-pd (checking-stack pushdown) machine, introduced in [38]. We now define the checking-stack controlled machine types. The three stages of computation are modelled by adding a finite component $\{\text{push, read, pop}\}$ to the configurations.

4.3. DEFINITION. Let $D = (S, s_0, S_\infty, I, m)$ be a machine type. The *checking-stack controlled machine type* corresponding to D , denoted by $CSP(D)$, is $(S'', s_0'', S''_\infty, I'', m'')$, where (using the concepts of Definition 4.1)

$$S'' = S' \times \{\text{push, read, pop}\},$$

$$s_0'' = (\lambda, \text{push}),$$

$$S''_\infty = \{(\lambda, \text{pop})\},$$

$$I'' = I, \text{ and}$$

m'' is the same as m' on the S' component of the configuration; for the second component, the $\text{push}(\gamma)$ instruction changes push into push and is undefined for read and pop, the $\text{movedown}(\gamma)$ instruction changes push and read into read and is undefined for pop, the moveup instruction changes read into read and is undefined for push and pop, and the pop instruction changes push, read, and pop into pop; the tests are independent of the second component, except that stackempty is true only for (λ, push) , for technical reasons. ■

Dropping the facility of extra pushdown squares from $SP(D)$ and $CSP(D)$ we obtain the corresponding machine types of Fig. 1, which will be denoted by $S(D)$ and $CS(D)$, respectively. The formal definitions are left to the reader; these machine types have one instruction movedown rather than $\text{movedown}(\gamma)$ for all $\gamma \in \Gamma$. We note that Lemma 4.2 holds for all these machines.

Clearly all machine types discussed above satisfy the requirement of Definition 1.1, that the set of successful instruction sequences is not empty: $\lambda \in L_{SP(D)}$ and $(\text{push}(\gamma); \text{pop}) \in L_{CS(D)}$. In fact they are even nontrivial (Definition 1.7): stackempty is a test on the initial configuration, and configuration different from the initial one can be reached by the instruction $\text{push}(\gamma)$ and then transformed into a final configuration by a pop .

We now show that if the underlying machine type is nontrivial, the extra pushdown squares in the $SP(D)$ machine type are superfluous.

4.4. LEMMA. *For every nontrivial machine type D , $K(SP(D)) = K(S(D))$ and $K(CSP(D)) = K(CS(D))$.*

Proof. Clearly every machine of type $S(D)$ can be simulated by a machine of type $SP(D)$ by using a dummy symbol on the pushdown squares. In the other direction, we will code the pushdown symbols into the D -tapes (this idea is from [26]). To be able to do this, D has to have at least two configurations (s_0 and s_1) between which it is able to distinguish and which it is able to install in and remove

from the storage. This property of the storage structure is precisely the notion of nontriviality (see the discussion following Definition 1.7). The simulation is now similar to the one in Lemma 4.2. A configuration of $SP(D)$ shown in (a) of Fig. 5 is simulated by a configuration of $S(D)$ as shown in (b) of Fig. 5 (assuming, by Lemma 4.2, that there are just two pdsymbols, say, 0 and 1). The stack symbol x is a dummy symbol; for $1 \leq i \leq n-1$, $r_i = s_0$ or s_1 if $\beta_i = 0$ or 1, respectively; r_0 is a dummy D -tape; β_n is kept in the finite control. The details of the simulation are left to the reader. ■

We finally arrive at the main result of this section: the correspondence between extended basic macro grammars and stack controlled machines.

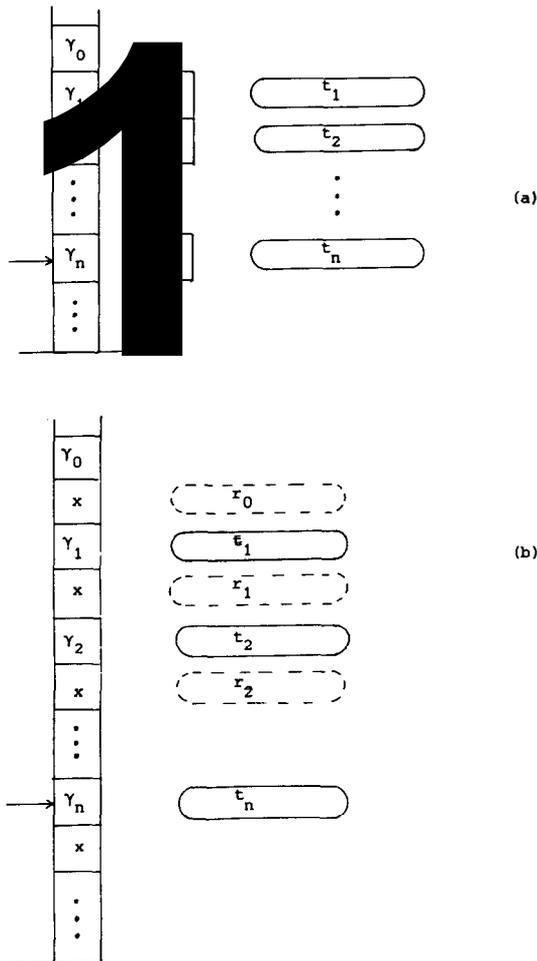


FIG. 5. $S(D)$ simulation of $SP(D)$.

4.5. THEOREM. *For every machine type D , $K(SP(D)) = B(K(D))$ and $K(CSP(D)) = LB(K(D)) = H(K(D))$.*

Proof. The proof is a direct generalization of the proof of equivalence of the (finitely) extended basic macro grammars and the s-pd machines in [15]. For more motivation we refer to that paper. The equality of $LB(K(D))$ and $H(K(D))$ follows from Theorem 1.5 and Proposition 2.2.

We first show the inclusions $B(K(D)) \subseteq K(SP(D))$ and $LB(K(D)) \subseteq K(CSP(D))$ by direct recognition of the generated languages by the machines. Consider a $K(D)$ -extended basic macro grammar $G = (F, \Psi, \Sigma, X, S, d, P)$ with $X = \{x_1, \dots, x_m\}$. For each language name $\psi \in \Psi_n$, let M_ψ be a machine of type D which accepts $d(\psi)$. By taking disjoint unions we may assume (to simplify notation) that these machines have the same set of states, the same set of final states, the same finite control δ , and the same finite subset I_1 of I . Thus, $M_\psi = (Q, \Sigma \cup \{x_1, \dots, x_m\}, I_1, q_\psi, Q_\infty, \delta)$; we also assume that if $(q, w, p, u) \in \delta$, then $w \in \Sigma \cup \{x_1, \dots, x_m\} \cup \{\lambda\}$.

The $SP(D)$ -machine M to recognize $L(G)$ uses all right-hand sides of productions in P (and their suffixes) as stack symbols. After nondeterministically simulating a left-most derivation with rules from P on the stack, it will eventually recognize a string generated by a language name ψ . To do this, M opens a D -tape to simulate a computation of M_ψ . Since the simulation of the derivation of G , in the stack, was merely symbolic, in order to determine the actual arguments of the language name ψ , M has to move down deeper into the stack, simulating more and more D -machines.

M will be specified by an “ $SP(D)$ -program” using the instruction set of $SP(D)$ in addition to some well-known programming constructs. It is left to the reader to implement this program as the finite control of M .

In the program we use the following notation. We use “stacksymbol” to denote the square pointed at by the stack pointer (viewed as a location); similarly for pdsymbol. We use a global location “state” (with finitely many values) to keep the current state of the D -machine simulated on the current D -tape. On the current pushdown square we store the state in which to resume the (interrupted) computation on the D -tape one square higher. The pdsymbol \$ is a dummy symbol. For $a \in \Sigma$, $read(a)$ means that a is read from the input (and the machine blocks if a is not the current input symbol); $read(\lambda)$ is an empty instruction. Each stack symbol t is of the form $t = t_1 t_2 \cdots t_k$ ($k \geq 0$), where t_i is either $\psi(\mathbf{x})$ or $A(\phi(\mathbf{x}))$; we shall denote t_1 by $head(t)$ and $t_2 \cdots t_k$ by $tail(t)$. We use **or** between two statements to indicate a nondeterministic choice between them (in the usual sense of automata theory). The program for the $SP(D)$ -machine is as follows:

```

begin push( $S$ );
cycle: if stacksymbol =  $\lambda$ 
    then pop; if stackempty then halt
        else stacksymbol := tail(stacksymbol) fi
    else case head(stacksymbol) of

```

```

    A(...): push(any right-hand side of a rule for A);
     $\psi(\dots)$ : begin state :=  $q_\psi$ ;
                movedown($);
                EVAL;
                stacksymbol := tail(stacksymbol)
                end
    esac fi;
goto cycle
end;

```

where EVAL denotes the following routine text:

```

begin
  loop:
  if (state,  $a$ ,  $q$ ,  $u$ )  $\in \delta$  with  $a \in \Sigma \cup \{\lambda\}$ 
    then state :=  $q$ ; read( $a$ );  $u$  fi
  or
  if (state,  $x_j$ ,  $q$ ,  $u$ )  $\in \delta$ 
    then let  $\psi(\dots)$  be th  $j$ th argument of head(stacksymbol);
         $u$ ; movedown( $q$ );
        state :=  $q_\psi$  fi
  or
  if state  $\in Q_\infty$ 
    then state := pdsymbol; moveup;
        if pdempty then return fi fi;
  goto loop
end EVAL.

```

Note that in the case-statement of the main program there should be a clause for each nonterminal $A \in F$ and each language name $\psi \in \mathcal{P}$. The EVAL routine really consists of a nondeterministic choice between a large number of statements, one for each element of δ and each element of Q_∞ .

It is left to the reader to show that the program works correctly. In the *LB*-case we first transform G according to Lemma 2.3 (note that, by Theorem 1.5, $K(D)$ satisfies the assumption of Lemma 2.3). Then the same program needs a checking-stack only.

We now turn to the second part of the proof. The inclusions $K(SP(D)) \subseteq B(K(D))$ and $K(CSP(D)) \subseteq LB(K(D))$ can be proved by showing, roughly speaking, that the full semi-AFL generators of $SP(D)$ and $CSP(D)$ can be generated by the corresponding type of macro grammars. In fact, by Theorem 1.5, $K(D)$ is full semi-AFL. Hence, by Theorem 3.6 and Proposition 2.2, $B(K(D))$ and $LB(K(D))$ are full semi-AFLs. Thus, by Theorem 1.5, it suffices to show that for every finite set I_1 of instructions, $L_{SP(D)} \cap (I_1)^* \in B(K(D))$ and $L_{CSP(D)} \cap (I_1)^* \in LB(K(D))$.

To show this we first reformulate $SP(D)$ and $CSP(D)$ to have a more convenient instruction set: in the notation of Definition 4.1, we now take I' to consist of I and

all instructions a , a^E , a_γ^D , a_γ^U with $a, \gamma \in \Gamma = \{0, 1\}$; see Lemma 4.2. The meaning of these instructions is, as in [15], as follows:

- a : push the symbol a on top of the stack;
- a^E : pop the symbol a off the stack (and block if the top symbol is not a);
- a_γ^D : movedown from the stacksymbol a , push γ on top of the pushdown, and open a new D -tape
- a_γ^U : close the D -tape, pop γ off the pushdown, and move up to the stacksymbol a .

Note that a and a^E are each others inverse, and so are a_γ^D and a_γ^U . It is easy to see that these new instructions can be simulated by the old instructions. It is left as an exercise for the reader to check that every machine using the old instructions can also be simulated by a machine using the new ones. We just note that if we keep the current pdsymbol in the finite control and put the pdsymbol of the D -tape one square higher in the current pdsquare (as we did, in fact, in the recognition program), then we only have to test pdsymbol when moving up (for which a_γ^U can be used); to test the value of the current stack symbol we can nondeterministically do $(a_\gamma^D; v; a_\gamma^U)$, for all γ , where v is an element of L_D .

We now have to show that for any finite subset I'_1 of I , $L_{SP(D)} \cap (I'_1)^* \in B(K(D))$. Let $I_1 = I'_1 \cap I$. Denote $L_D \cap I_1^*$ by L , and let L_x denote $\{xi_1xi_2x \cdots xi_nx \mid i_1i_2 \cdots i_n \in L\}$, where x is a new symbol. Clearly $L_x \in K(D)$. Now $L_{SP(D)} \cap (I_1)^*$ is generated by the extended basic macro grammar with the following productions:

$$\begin{aligned} S &\rightarrow aB(\lambda) a^E S && \text{for } a \in \{0, 1\}, \\ S &\rightarrow \lambda \\ B(x) &\rightarrow xaB(\psi_a(x)) a^E B(x) && \text{for } a \in \{0, 1\}, \\ B(x) &\rightarrow x. \end{aligned}$$

Language names with finite domains are indicated by their domains; $d(\psi_a) = (a_0^D L_x a_0^U \cup a_1^D L_x a_1^U)^*$. Clearly these domains are in $\text{REG} \leftarrow K(D)$. Hence $L_{SP(D)} \cap (I_1)^*$ is in $B(\text{REG} \leftarrow K(D))$. Since $B(\text{REG} \leftarrow K(D)) \subseteq B(K(D))$ by Theorem 1.5, Proposition 2.2, and Theorem 3.1, the result follows.

In the above grammar, the language parameter x stands for the set of all sequences of instructions a_γ^D and a_γ^U which can be executed on a certain stack s_x , starting and ending at the top of s_x . $B(x)$ generates the set of all instruction sequences that can be executed starting and ending with s_x without changing its contents in the intermediate steps. In the third rule above the symbol a is put on top of the stack s_x . Thus the new argument of B should consist of all instruction sequences which (repeatedly) first move down one square (opening a D -tape), then do an instruction sequence of the old argument x of B mixed (at the appropriate places) with instructions on the just-opened D -tape (of which x knew nothing!), and finally move up

again. This is modelled by the domain of ψ_a . It is left to the reader to formalize these statements and to prove the grammar correct.

In the $CSP(D)$ case $L_{CSP(D)} \cap (I_1)^*$ is generated by the extended linear basic macro grammar with productions

$$\begin{aligned} S &\rightarrow aB(\lambda) a^E && \text{for } a \in \{0, 1\}, \\ B(x) &\rightarrow aB(\psi_a(x)) a^E && \text{for } a \in \{0, 1\}, \\ B(x) &\rightarrow x, \end{aligned}$$

where $d(\psi_a)$ is the same as above. Hence the language is in $LB(REG \leftarrow K(D)) \subseteq LB(K(D))$. This proves the theorem. ■

4.6. COROLLARY. *For every nontrivial machine type D , $K(S(D)) = B(K(D))$ and $K(CS(D)) = LB(K(D))$.*

Proof. Immediate from Theorem 4.5 and Lemma 4.4. ■

Note that there are trivial machine types for which the equalities in Corollary 4.6 do not hold. In fact, if D is any machine type not satisfying condition (2) of Definition 1.7, then $S(D)$ is the stack automaton type and $K(D) = REG$. Hence $K(S(D)) = Stack \not\subseteq EB = B(K(D))$ and $K(CS(D)) = CStack \not\subseteq ETOL = LB(K(D))$, cf. [15].

4.7. COROLLARY. *For every machine type D , $K(CSP(D))$ is the smallest full hyper-AFL containing $K(D)$.*

Proof. Immediate from Theorem 4.5 and Proposition 2.2. ■

Thus we obtain the result that the checking-stack controlled machine types characterize the full hyper-AFLs (just as all machine types characterize the full semi-AFLs, see Theorem 1.6). See also [33].

4.8. THEOREM. *Let K be a class of languages. Then the following statements are equivalent:*

- (i) K is a full hyper-AFL.
- (ii) $K = K(CS(D))$ for some nontrivial machine type D .
- (iii) $K = K(CSP(D))$ for some machine type D .

Proof. (ii) \Rightarrow (iii) by Lemma 4.4.

(iii) \Rightarrow (i) by Corollary 4.7.

(i) \Rightarrow (ii). By Theorem 1.6 and a remark following Definition 1.7, there is a non-trivial machine type D such that $K = K(D)$. By Corollary 4.6, $K(CS(D)) = LB(K(D)) = LB(K)$. Since K is a full hyper-AFL, $LB(K) = K$. So $K(CS(D)) = K$. ■

The same result holds for full principal hyper-AFLs and finitely encoded checking-stack controlled machine types.

4.9. COROLLARY. *If K is a full principal semi-AFL, then so are $H(K)$ and $B(K)$.*

Proof. By Theorem 1.6, there exists a finitely encoded machine type D such that $K = K(D)$. Hence, by Lemma 4.2 (and its analog for CSP), $CSP(D)$ and $SP(D)$ are also finitely encoded. By Theorem 1.6 again, $K(CSP(D))$ and $K(SP(D))$ are full principal semi-AFLs. The result now follows from Theorem 4.5. ■

We make the following two observations:

Theorem 4.5 implies that Theorem 3.1 can now be reformulated as a result on machines; for every nontrivial machine type D , $S(CS(D))$ and $S(D)$ are equivalent machine types. It would not be difficult to prove this directly for machines. In fact, during a given computation of an $S(CS(D))$ -machine, only finitely many $CS(D)$ -tapes are opened (and closed) for any given stack square. Thus an $S(D)$ -machine could simulate this by guessing the contents of the corresponding checking stacks (after the push-stage), generating them on the stack immediately above the stack symbol. When moving down to the stack symbol, the $S(D)$ -machine nondeterministically picks one of these contents to simulate the behavior of the $S(CS(D))$ -machine on the corresponding $CS(D)$ -tape. Note that this corresponds closely to the grammatical construction given in the proof of Theorem 3.1.

In [13] it is shown that $B^*(\text{FIN})$ is the union of the infinite hierarchy of full hyper-AFLs $H(B^n(\text{FIN}))$, $n \geq 0$ (see also Proposition 5.7 in the next section; in fact, Theorem 4.5 of [13] states that $B^n(\text{FIN}) \subsetneq H(B^n(\text{FIN})) \subsetneq B^{n+1}(\text{FIN})$ for all n). By our Theorem 4.5 we now obtain concrete machine models for these concrete full hyper-AFLs, viz., $CS(S^n(D_R))$ (where D_R is some nontrivial machine type for REG): a checking-stack controlled iteration of stack controlled machines. In the next section we show the relationship between these machine types and nested stack automata with bounded depth of nesting.

5. BOUNDED NESTED STACK MACHINES

The way in which the stack controlled machine behaves is very similar to the behavior of the nested stack automaton of [2]. If, e.g., PD is the (usual) pushdown machine type, then $S(PD)$ corresponds closely to a (restricted) nested stack automaton in which just pushdown tapes are nested in the main stack. In particular, cf. Fig. 1, we could nest each pushdown just above the square it is associated with (in this way the nested stack automaton can reach both the current stack square and the current PD -tape). When iterating the operation $S(D)$ on machine types (starting with, say, a machine type for the regular languages) we would get machines closely related to the nested stack automaton with a fixed bound on the depth of nesting. On the other hand, by Theorem 4.5, the

corresponding class of languages would be $B^*(REG)$, the smallest full basic-AFL (cf. the end of Sect. 3).

In this section we define a nested-stack controlled machine type $NS(D)$ corresponding to any machine type D , and show that its bounded version (the bounded nested-stack controlled machine type $BNS(D)$) gives a machine characterization of the full basic-AFLs. In particular, the class BNS of (ordinary) bounded nested stack automata defines the smallest full basic-AFL.

We first give an informal description of (a very slight variation of) the nested stack automaton of [2]. A *nested stack* is a finite collection of stacks such that each of its stacks except one (the main stack) is nested between two squares of another stack in the collection (and there are no circularities in the nesting relation); moreover a nested stack has one stack pointer which points at one of its stacks and (if that stack is not empty) at one of the squares of the stack.

In a picture we will indicate the nesting of stacks as in Fig. 6. Stack s_2 is nested between squares a and b of stack s_1 ; this is indicated by connecting square b of s_1 with the bottom square c of stack s_2 . For the nested stack machine this means that the stack pointer can move from b to c and vice versa (via the connecting branch), but cannot move between a and b any more. (In [2] s_2 is actually put between squares a and b surrounded by endmarkers). An empty stack will be indicated by a circle (thus the stack pointer may point to a circle). If s_2 in Fig. 6 is empty, then there is a connection of b with the circle s_2 . An example of a nested stack (with 8 stacks) is given in Fig. 7; the arrow is the stack pointer (disregard the shading of the squares for the moment). Note that, due to the noncircularity of the nesting relation, the nested stacks may be viewed as a tree of stacks (see [17]).

With every stack of a nested stack we associate a *level number* which indicates its depth of nesting in the main stack. The level of the main stack is 1, and, if s_2 is nested in s_1 , then the level of s_2 is one plus the level of s_1 . In Fig. 7 the maximal level is 4.

5.1. DEFINITION. The *nested-stack machine type* NS is $(N, n_0, N_\infty, I_{NS}, m_{NS})$, where N is the set of all nested stacks (with symbols from Γ), n_0 is the empty nested stack (consisting of one empty stack), $N_\infty = \{n_0\}$ and I_{NS} consists of all instructions

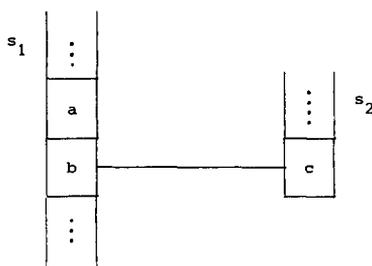


FIG. 6. Nesting of stack s_2 in stack s_1 .

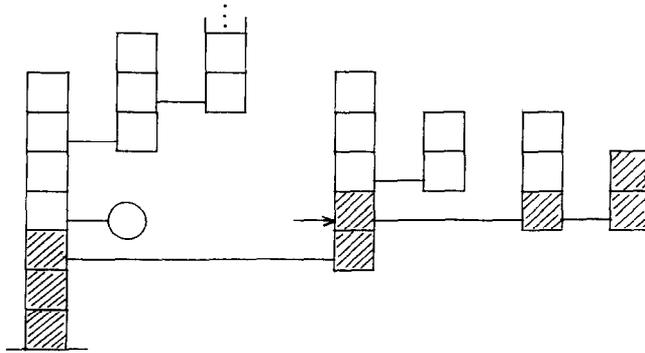


FIG. 7. A nested stack.

push(γ), pop, create, destruct, movedown, moveup, and all tests $\text{stacksymbol} = \gamma$, topstack , stackempty (for $\gamma \in \Gamma$). The meaning m_{NS} of the instructions (closely following [2]) is illustrated in Figs. 8–10, where only the “local situation” is shown. Figure 8 shows that $\text{push}(\gamma)$ and pop are the usual operations, changing the top of the stack pointed at. Figure 9 shows that, at square a , an empty stack can be created, nested between a and b ; moreover, such a stack can be destructed (only) if it is empty. Figure 10 shows that “movedown” moves the stack pointer down in the same stack (provided it does not pass a nested stack) or, if it is at the bottom of the stack (or the stack is empty), moves it to the “mother stack.” The same holds, vice versa, for “moveup.” Thus, in terms of the pictures, “moveup” means to move east if possible and move north otherwise; “movedown” means to move east if possible and move south otherwise (cf. [17]). Finally, the test $\text{stacksymbol} = \gamma$ is true iff the stack pointer points to a square containing γ , topstack is true iff the stack pointer points to the top of a stack, and stackempty is true iff the stack pointer points at an empty stack. ■

Note that the top square of a stack is never connected to another stack. Note also that if a nested stack machine never does a “wrong” moveup (Fig. 10(i) with a stack nested between a and b), i.e., always prefers east to north, then it can never make a “wrong” movedown, i.e., it can freely go south. This means that the stack

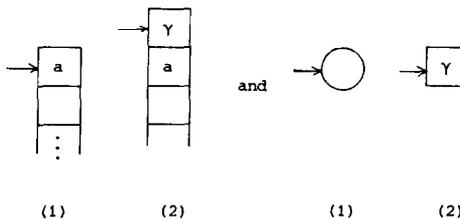


FIG. 8. $\text{push}(\gamma)$: (1) \Rightarrow (2); pop : (2) \Rightarrow (1).

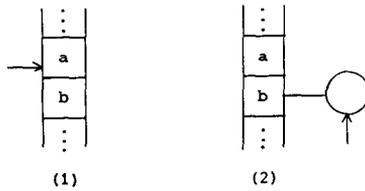


FIG. 9. create: (1) \Rightarrow (2); destruct: (2) \Rightarrow (1).

pointer can only be in the shaded area of Fig. 7. A formal proof of this is left to the reader.

The (trivial) differences between Definition 5.1 and the nested stack automaton of [2] are that in [2]: stacks are surrounded by endmarkers ϵ and $\$$; any number of symbols can be pushed in one move; any given stack can be created in one move (rather than just the empty stack); stacks can be nested below any symbol rather than between any two symbols (i.e., there may be an additional stack nested below the bottom of a stack).

We now generalize the nested stack automaton by allowing it to create and destruct D -tapes. Let, for a given machine type D , $N(D)$ be the set of "nested stacks with D -tapes," obtained by allowing configurations of D to be nested between squares, i.e., allowing s_2 to be a D -configuration in Fig. 6; see Fig. 11 and compare it to Fig. 1. The stack pointer may now also point at a D -tape, and each D -tape in a nested stack has a level number.

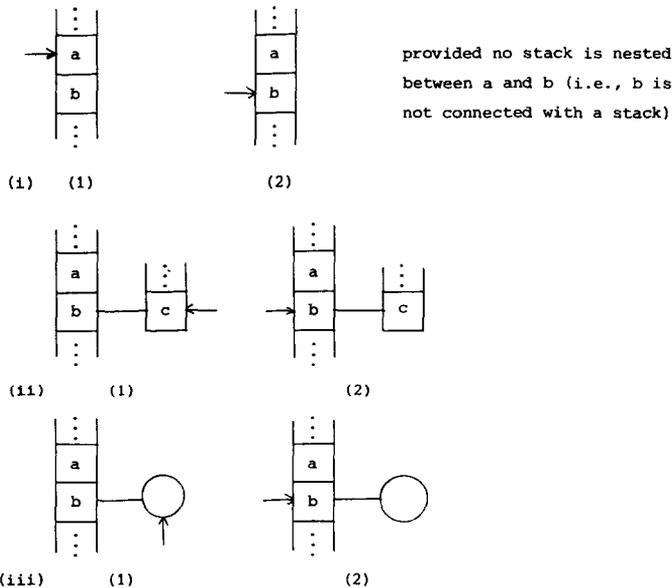


FIG. 10. movedown: (1) \Rightarrow (2); moveup: (2) \Rightarrow (1).

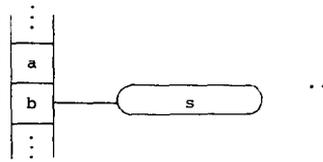


FIG. 11. A D -tape s nested between a and b .

5.2. DEFINITION. For every machine type $D = (S, s_0, S_\infty, I, m)$, the *nested-stack controlled machine type* $NS(D)$ corresponding to D is $(N(D), n_0, \{n_0\}, I_{NS(D)}, m_{NS(D)})$, where n_0 is the empty nested stack. The set of instructions $I_{NS(D)}$ is $I \cup I_{NS} \cup \{\text{create}(D), \text{destruct}(D), D\text{-tape}\}$. The meaning of the tests in I_{NS} and the instructions $\text{push}(\gamma)$, pop , create , and destruct is the same as for NS . The meaning of $\text{create}(D)$ and $\text{destruct}(D)$ is the same as in Fig. 9, provided the circle represents the initial configuration s_0 of D for $\text{create}(D)$, and some final configuration in S_∞ for $\text{destruct}(D)$. The meaning of movedown and moveup is as in Fig. 10, where in Fig. 10(i) no stack or D -tape may be nested between a and b , and in Fig. 10(iii) the circle may also represent any element of S . For $i \in I$, the meaning of i is defined (only) if the stack pointer points at a D -tape, and in that case $m(i)$ is applied to that D -tape. Finally, the test “ D -tape” is true iff the stack pointer points at a D -tape. ■

We observe that one could define K -extended macro grammars (not necessarily basic), as in [6], and show that these correspond to the $NS(D)$ -machines (generalizing the result of [18, 2] that $K(NS) = \text{OI}$). Here instead we consider only the case of bounded depth of nesting. We say that an element n of N or $N(D)$ has *depth of nesting* k if k is the maximal level of the stacks and D -tapes of n .

5.3. DEFINITION. A machine M of type $NS(D)$, or NS , is *bounded* with bound k if for each string $w \in L(M)$ there exists a computation of M accepting w such that every configuration in that computation has depth of nesting at most k .

We denote by $BNS(D)$, or BNS , the class of bounded machines (for any bound ≥ 1) of type $NS(D)$, or NS , respectively. Formally, we can define the *bns-controlled machine type* $BNS(D)$, with the same configurations as $NS(D)$, which has the (infinite) instruction set $I_{NS(D)} \times \{1, 2, 3, \dots\}$, i.e., the instructions of $NS(D)$ together with an index k which indicates the depth of nesting. The meaning of instruction (i, k) is the same as that of i , except that it is undefined if $m_{NS(D)}(i)$ would produce a nested stack with depth of nesting $> k$. Clearly, every bounded $NS(D)$ -machine (with bound k) can be turned into a machine of type $BNS(D)$ by adding index k to all instructions it uses. Vice versa, a $BNS(D)$ -machine M can be simulated by a bounded $NS(D)$ -machine by the following basic trick. Since M uses only finitely many instructions, the maximal index k of these instructions exists. Construct a new $BNS(D)$ -machine M' which on every square (on a second track) prints the level number of the stack it belongs to, up to k . Since M' can now be changed such that

it checks the depth of nesting itself before using an instruction, the indices can be dropped from the instructions and an $NS(D)$ -machine is obtained which is bounded with bound k . The same can be done for BNS . Hence $K(BNS(D))$ is the class of languages accepted by bounded $NS(D)$ -machines, and similarly $K(BNS)$ is the class of languages accepted by bounded nested stack machines.

We now turn to the main result of this section. Recall that $B^*(K)$ is the union of all $B^k(K)$, $k \geq 0$ (Definition 3.10). For every full semi-AFL K , $B^*(K)$ is the smallest full basic-AFL containing K .

5.4. THEOREM. *For every nontrivial machine type D , $K(BNS(D)) = B^*(K(D))$: the smallest full basic-AFL containing $K(D)$.*

Proof. Since, by Theorem 4.5 and Corollary 4.6, $K(SP^k(D)) = K(S^k(D)) = B^k(K(D))$, it suffices to show that $\bigcup \{K(S^k(D)) \mid k \geq 1\} \subseteq K(BNS(D)) \subseteq \bigcup \{K(SP^k(D)) \mid k \geq 1\}$, where $SP^1(D) = SP(D)$ and $SP^{k+1}(D) = SP(SP^k(D))$, and similarly for $S^k(D)$. We will prove this by showing how these machines can simulate each other. Before doing this, let us consider for a moment the machine type $S^k(D)$. Clearly, cf. Figs. 1 and 11, we may view each of its configurations as an element of $N(D)$, i.e., a nested stack with D -tapes, except that each of its stacks has a private stack pointer. Note also that all stacks have level number $\leq k$ whereas all D -tapes have level number $k + 1$. Furthermore, the set of instructions of $S^k(D)$ consists of the set I of instructions of D (working on level $k + 1$) and instructions $\text{push}(\gamma)$, pop , movedown , moveup , stackempty , pdempty , $\text{stacksymbol} = \gamma$ at each level, i.e., a different set of these instructions for each level from 1 to k (due to the renaming convention in Definition 4.1). At each level of the nested stack there is precisely one stack (or D -tape) which can be accessed by the instructions of that level (cf. the stacks with shaded squares in Fig. 7). Note that on levels 1 to $k - 1$ the instruction "movedown" creates an empty stack, whereas on level k it opens a D -tape. (Note also that there is at most one level on which the $\text{push}(\gamma)$ and pop instructions are successful). The machine type $SP^k(D)$ is similar to $S^k(D)$; it just has an extra pushdown square "on each nesting connection", i.e., "on" each line connecting two stacks or a stack with a D -tape (Figs. 6 and 11). We are now ready to discuss the simulations.

We first show that $K(BNS(D)) \subseteq \bigcup \{K(SP^k(D)) \mid k \geq 1\}$. Let M_1 be a bounded $NS(D)$ -machine with bound k . First we change M_1 in such a way that its bound is $k + 1$ and its nested stacks have stacks with level number $\leq k$ and D -tapes with level number $k + 1$, as follows. First use the trick discussed above of printing the level number on each square. Now, if M_1 wants to create a new D -tape (by create (D)), it should instead create a sequence of small (dummy) stacks to increase the level number to k , and then create the D -tape at level $k + 1$ (to create the dummy stacks use the instruction sequence create ; $\text{push}(\gamma)$; $\text{push}(\gamma)$, repeatedly). In the rest of the computation M_1 should of course always take these dummy stacks into account. The details of this construction are left to the reader.

We now show how M_1 can be simulated by an $SP^k(D)$ -machine M_2 . For each

instruction of M_1 we will give a piece of program with which the instruction is simulated by M_2 ; as usual it is left to the reader to implement this with a finite control δ . M_2 uses a location n (initialized to 1) in its finite control to store the current level of the stack pointer of M_1 . We assume that M_2 has a test "bottomstack" at each level. There is one additional problem: in the situation of Fig. 10(i), M_2 can only simulate M_1 by actually nesting a (dummy) stack or D -tape between a and b . Thus, when moving up, M_2 has to know for each square whether or not the stack (or D -tape) it is connected to is a dummy. This will be handled by printing "yes" or "no" on the corresponding pdsquare. We will use the statement "finalize on level n " to bring the current (dummy) tape on level n into a final configuration.

The simulation is as follows:

| | |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $M_1: NS(D)$ | $M_2: SP^k(D)$ |
| $i \in I$ | if $n = k + 1$ then i else block fi |
| push(γ) | push(γ) on level n |
| pop | pop on level n |
| create, create(D) | movedown(yes) on level n ; $n := n + 1$ |
| destruct, destruct(D) | $n := n - 1$; moveup on level n |
| movedown | if bottomstack on level n or stackempty on level n or $n = k + 1$ then $n := n - 1$ else movedown(no) on level n fi |
| moveup | if pdsymbol = yes on level n then $n := n + 1$ else finalize on level n ; moveup on level n fi |
| stacksymbol = γ | stacksymbol = γ on level n |
| topstack | pdempty on level n |
| stackempty | stackempty on level n |
| D -tape | $n = k + 1$ |

To understand the correctness of the simulation the reader should keep in mind the positions of the stack pointers of M_2 . The stack pointer of M_1 is simulated by the stack pointer of M_2 in the same stack, say, s . The stack pointer of M_2 in the mother of s (i.e., the stack in which s is nested) points at the square which is connected with the bottom of s ; the same holds for the mother of s (and her mother), and so on, up to the main stack. For all other stacks, the stack pointer of M_2 is at the bottom (or the stack is empty).

We now turn to the simulation in the other direction. Let M_1 be an $S^k(D)$ -machine. We will show that M_1 can be simulated by a bounded $NS(D)$ -machine M_2 with bound $k + 1$. First we change M_1 in such a way that it prints the level number

in each square of its stacks (with every push move). M_2 will simulate M_1 with exactly the same configurations, except that it has just one stack pointer. To find the current position of the stack pointer of M_1 at a given level it uses the following routine "move to level n ," which is based on the fact that M_1 always creates a stack or a D -tape when moving down (i.e., its nested stacks look like Fig. 1 on each level). Thus, in that routine M_2 first moves up to the highest level and then moves down to the first square of level n .

The routine "move to level n " is as follows ($n \leq k + 1$):

```

begin
  while not ( $D$ -tape or stackempty) do moveup od;
  if  $D$ -tape then level :=  $k + 1$ 
    else movedown;
    level := 1 + level of stacksymbol;
    moveup fi;
  if level <  $n$  then block
  else if level >  $n$ 
    then repeat movedown until  $n =$  level of stacksymbol end
    { else level =  $n$  } fi fi
end move to level  $n$ .

```

For each instruction of M_1 (on each level) we now give the corresponding simulation by M_2 :

| | |
|-------------------------------------|--------------------------------------|
| $M_1: S^k(D)$ | $M_2: NS(D)$ |
| $i(\in I)$ | move to level $k + 1; i$ |
| push(γ) on level n | move to level n ; push(γ) |
| pop on level n | move to level n ; pop |
| movedown on level n | move to level n ; |
| | if $n = k$ then create(D) |
| | else create fi; |
| | movedown |
| moveup on level n | move to level n ; |
| | moveup; |
| | if $n = k$ then destruct(D) |
| | else destruct fi |
| stackempty on level n | move to level n ; stackempty |
| pempty on level n | move to level n ; topstack |
| stacksymbol = γ on level n | move to level n ; |
| | stacksymbol = γ |

This ends the description of the simulation of M_1 by M_2 , and the theorem is proved. ■

We can now state the characterization of the full basic-AFLs by the BNS -controlled machine types.

5.5. THEOREM. *Let K be a class of languages. K is a full basic-AFL if and only if $K = K(BNS(D))$ for some nontrivial machine type D .*

Proof. Immediate from Theorem 5.4 and Theorem 1.6. ■

Finally we show that $K(BNS)$ is the smallest full basic-AFL. Thus, together with Theorem 3.12, we have machines and grammars for this class.

5.6. THEOREM. $K(BNS) = B^*(FIN)$.

Proof. Take the nontrivial machine type $D_R = (S, s_0, S_\infty, I, m)$ with $S = S_\infty = \{s_0, s_1\}$, $I = \{0, 1, t_0, \bar{t}_0\}$ with $m(0) = \{(s_1, s_0)\}$, $m(1) = \{(s_0, s_1)\}$, and t_0 and \bar{t}_0 are a test on s_0 . Clearly $K(D_R) = REG$. Hence, by Theorem 5.4, $K(BNS(D_R)) = B^*(REG) = B^*(FIN)$. It is easy to see that $K(BNS(D_R)) = K(BNS)$. ■

Note that since $K(BNS)$ is the smallest full basic-AFL, BNS is the “smallest” nontrivial machine type D (i.e., smallest with respect to $K(D)$) such that $K(D) = K(S(D))$. Thus, informally, BNS is the smallest machine type D such that $D = S(D)$, i.e., a stack controlled machine type of which the stack controls its own machine type (of course, any other full basic-AFL, such as $K(NS)$, also has that property).

It follows from the proof of Theorem 5.4 that $SP^k(D)$ is equivalent to $BNS(D)$ -machines with bound $k+1$, that have all their D -tapes at level $k+1$ (and no stacks on that level). In particular, if D_R is the nontrivial machine type of the proof of Theorem 5.6, then we obtain that $B^k(FIN)$ is accepted by BNS -machines with bound $k+1$, which at level $k+1$ have stacks of length one only. It will be shown in [13] that $\{B^k(FIN)\}_{k \geq 1}$ is a proper hierarchy, i.e., $B^k(FIN) \subsetneq B^{k+1}(FIN)$ for every $k \geq 1$. Let BNS_k denote the class of bounded nested stack automata with bound k . It now follows that $\{BNS_k\}_{k \geq 1}$ is an infinite hierarchy. In fact, $K(BNS_k) \subseteq B^k(FIN) \subsetneq B^{k+1}(FIN) \subseteq K(BNS_{k+2})$. It is open whether $K(BNS_k) \subsetneq K(BNS_{k+1})$ for all $k \geq 1$.

We state the above in the following proposition.

5.7. PROPOSITION. *The bounded nested stack automata form an infinite hierarchy with respect to depth of nesting. In particular, $K(BNS_k) \subsetneq K(BNS_{k+2})$ for every $k \geq 1$.*

Let us finally see what happens if we nest checking-stacks, i.e., each stack of the nested stack has three stages: pushing, reading, and popping (where each stage may be interrupted several times). Let $NCS(D)$ denote the class of $NS(D)$ -machines which use checking-stacks, and let $BNCS(D)$ be the bounded version, and similarly for NCS and $BNCS$.

It is easy to check that Theorems 5.4 and 5.6 (and 5.5) also hold for the checking-stack/linear case: $K(BNCS(D)) = LB^*(K(D))$. Hence by Proposition 2.2 and Theorem 4.5, $K(BNCS(D)) = LB(K(D)) = K(CS(D))$. Thus bounded nesting of checking-stacks has the same power as just one checking-stack. In particular, the bounded nested checking-stack automata have the same power as the cs-pd

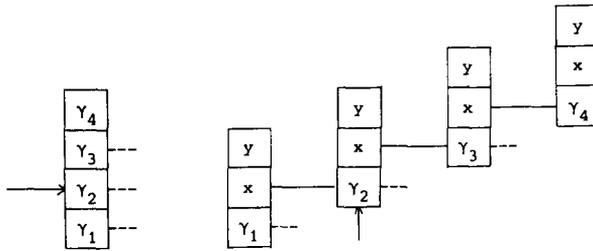


FIG. 12. Simulation of a stack by nested checking-stacks.

machines of [38, 15], i.e., they accept the class ETOL. This is in contrast with the fact that the (unbounded) nested checking-stack automaton has the same power as the nested stack automaton (also with D -tapes). In fact, to simulate a nested stack automaton M_1 by a nested checking-stack automaton M_2 (in which, even, every checking-stack has length 3) we replace a stack $\gamma_1\gamma_2 \cdots \gamma_n$ by n checking-stacks of length 3 as indicated in Fig. 12 (it is just like the simulation of an arbitrary tree by a binary tree). The dashed lines indicate the possible connections to other stacks. The symbols x and y are “new” symbols. We assume that before and after each simulation of an instruction of M_1 the nonempty checking-stacks are in the reading stage (hence only reading and popping are safe). The empty stack is simulated by the empty stack (in the pushing stage) except that M_2 uses one extra checking-stack with bottom symbol $\#$ as new bottom of the main stack. M_2 starts the simulation of M_1 with **begin** push($\#$); push(x); push(y); movedown **end** and ends it with **begin** moveup; moveup; pop; pop; pop **end**.

M_2 simulates the instructions of M_1 as follows:

| | |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| M_1 : NS | M_2 : NCS |
| push(γ) | if not stackempty then moveup; moveup; create fi ; push(γ); push(x); push(y); movedown; movedown |
| pop | moveup; moveup; pop; pop; pop; destruct; if stacksymbol = y then movedown; movedown else {stacksymbol = x } create fi |
| movedown | movedown; if stacksymbol = x then movedown fi ; if stacksymbol = $\#$ then block fi |
| moveup | moveup; if stacksymbol = x then moveup fi ; if stacksymbol = y then block fi |
| create | movedown; create |
| destruct | destruct; moveup |

| | |
|------------------------|-------------------------------------|
| stacksymbol = γ | stacksymbol = γ |
| stackempty | stackempty or stacksymbol = $\#$ |
| topstack | moveup; moveup; stacksymbol = y ; |
| | movedown; movedown |

For $NS(D)$, the instructions $create(D)$ and $destruct(D)$ are simulated in the same way as $create$ and $destruct$ above, and the test “ D -tape” and all instructions $i \in I$ remain the same.

Thus we have shown

5.8. THEOREM. (i) *For every machine type D , $K(NCS(D)) = K(NS(D))$. In particular, $K(NCS) = K(NS)$.*

(ii) *For every nontrivial machine type D , $K(BNCS(D)) = K(CS(D)) = LB(K(D))$. In particular, $K(BNCS) = LB(FIN)$.*

CONCLUSION

We have generalized the notion of iterated substitution to that of basic substitution, by way of K -extended basic macro grammars. We have investigated the class $B(K)$ of languages obtained by applying all basic substitutions to the languages in K , in particular with respect to closure properties. It turned out that the classes $B(K)$ can be characterized by stack controlled machines, and the classes $LB(K)$, i.e., the full hyper-AFLs, by checking-stack controlled machines. Finally, the full basic-AFLs, i.e., full AFLs closed under basic substitution, are characterized by bounded nested-stack controlled machines. These results give more insight into the structure of the class OI of (outside-in) macro languages, and also in the precise relationship between restricted classes of macro grammars and restricted classes of nested stack automata. As far as we can see now, all interesting full basic-AFLs are contained in OI.

Finally, we hope to have shown in Sections 4 and 5 that the idea of “programming” automata [35, 30], which was used in [15] to study s-pd machines, not only has didactic advantages in courses on languages and automata, but leads to more readable proofs even in abstract automata theory.

ACKNOWLEDGMENT

We are grateful to Peter Asveld for many useful discussions. We wish to thank Ria Dirks for her excellent typing of the manuscript.

REFERENCES

1. A. V. AHO, Indexed grammars—An extension of context-free grammars, *J. Assoc. Comput. Mach.* **15** (1968), 647–671.
2. A. V. AHO, Nested stack automata, *J. Assoc. Comput. Mach.* **16** (1969), 383–406.
3. A. ARNOLD AND M. DAUCHET, Transductions de forêts reconnaissables monadiques; forêts corégulières, *Rev. Française Automat. Informat. Rech. Oper.* **IT-10** (1976), 5–28.
4. P. R. J. ASVELD, Controlled iteration grammars and full hyper-AFLs, *Inform. and Control* **34** (1977), 248–269.
5. P. R. J. ASVELD, “Iterated Context-independent Rewriting—An Algebraic Approach to Families of Languages,” Ph.D. thesis, Twente University of Technology, May 1978.
6. P. R. J. ASVELD, Generalizations of context-free grammars and the nonself-embedding property, manuscript, 1980.
7. P. R. J. ASVELD AND J. ENGELFRIET, Extended linear macro grammars, iteration grammars, and register programs, *Acta Inform.* **11** (1979), 259–285.
8. J. BERSTEL, “Transductions and Context-free Languages,” Teubner, Stuttgart, 1979.
9. P. J. DOWNEY, “Formal Languages and Recursion Schemes,” Ph.D. thesis, Report No. TR 16–74, Harvard University, 1974.
10. A. EHRENFEUCHT, G. ROZENBERG, AND S. SKYUM, A relationship between ETOL languages and EDTOL languages, *Theoret. Comput. Sci.* **1** (1976), 325–330.
11. S. EILENBERG, “Automata, Languages, and Machines,” Vol. A. Academic Press, New York, 1974.
12. J. ENGELFRIET, Macro grammars, Lindenmayer systems, and other copying devices, in “Automata, Languages, and Programming, 4th Colloq.” (A. Salomaa and M. Steinby, Eds.), Lecture Notes in Computer Science No. 52, pp. 221–229, Springer-Verlag, Berlin, 1977.
13. J. ENGELFRIET, Hierarchies of hyper-AFLs, Memorandum INF-82-3, Twente University of Technology, May 1982; *J. Comput. System Sci.*, in press.
14. J. ENGELFRIET AND E. MEINECHE SCHMIDT, IO and OI, I, *J. Comput. System Sci.* **15** (1977), 328–353; II, *J. Comput. System Sci.* **16** (1978), 67–99.
15. J. ENGELFRIET, E. MEINECHE-SCHMIDT, AND J. VAN LEEUWEN, Stack machines and classes of non-nested macro languages, *J. Assoc. Comput. Mach.* **27** (1980), 96–117.
16. J. ENGELFRIET AND G. SLUTZKI, Bounded nesting in macro grammars, *Inform. and Control* **42** (1979), 157–193.
17. G. FILÉ, “The Characterization of Some Families of Languages by Classes of Indexed Grammars,” MSc thesis, Pennsylvania State University, June 1977.
18. M. J. FISCHER, “Grammars with Macro-like Productions,” PhD thesis, Harvard University, 1968 (see also 9th Annual Symposium on Switching and Automata Theory, pp. 131–142).
19. G. A. FISHER AND G. N. RANEY, On the representation of formal languages using automata on networks, in “Proc. 10th Ann. Sympos. on Switching and Automata Theory, 1969,” pp. 157–165.
20. S. GINSBURG, “Algebraic and Automata-theoretic Properties of Formal Languages,” North-Holland, Amsterdam, 1975.
21. S. GINSBURG AND S. A. GREIBACH, Abstract families of languages, in “Studies in Abstract Families of Languages,” *Memoirs of the American Mathematical Society* No. 87, pp. 1–32, 1969.
22. S. GINSBURG, S. A. GREIBACH, AND M. A. HARRISON, One-way stack automata, *J. Assoc. Comput. Mach.* **14** (1967), 389–418.
23. J. GOLDSTINE, Automata with data storage, in “Proc. of A Conference on Theoretical Computer Science, Waterloo, 1977,” pp. 239–246.
24. J. GOLDSTINE, A rational theory of AFLs, “6th Colloq. on Automata, Languages, and Programming, Graz,” Lecture Notes in Computer Science No. 71, pp. 271–281, Springer-Verlag, Berlin, 1979.
25. J. GOLDSTINE, Formal languages and their relation to automata: what Hopcroft & Ullman didn’t tell us, in “Formal Languages: Perspectives and Open Problems” (R. V. Book, Ed.), Academic Press, New York, 1980.
26. S. A. GREIBACH, Full AFLs and nested iterated substitution, *Inform. and Control* **16** (1970), 7–35.

27. S. A. GREIBACH, Checking automata and one-way stack languages, *J. Comput. System Sci.* **3** (1969), 196–217.
28. S. A. GREIBACH, Chains of full AFLs, *Math. Systems Theory* **4** (1970), 231–242.
29. J. E. HOPCROFT AND J. D. ULLMAN, An approach to a unified theory of automata, *Bell System Techn. J.* **46** (1967), 1793–1829.
30. D. E. KNUTH AND R. H. BIGELOW, Programming languages for automata, *J. Assoc. Comput. Mach.* **14** (1967), 615–635.
31. M. NIVAT, On the interpretation of recursive program schemes, *Symposia Mat.* **15** (1975), 255–281.
32. G. ROZENBERG AND A. SALOMAA, “The Mathematical Theory of L Systems,” Academic Press, New York, 1980.
33. G. ROZENBERG AND D. VERMEIR, On acceptors of iteration languages, *Internat. J. Comput. Math.* **7** (1979), 3–19.
34. A. SALOMAA, Macros, iterated substitution and Lindenmayer AFLs, DAIMI PB-18, Aarhus University, 1973; abstract in “L Systems” (G. Rozenberg and A. Salomaa, Eds.), *Lecture Notes in Computer Science* No. 15, pp. 250–253, Springer-Verlag, Berlin, 1974.
35. D. SCOTT, Some definitional suggestions for automata theory, *J. Comput. System Sci.* **1** (1967), 187–212.
36. J. VAN LEEUWEN, *F*-iteration languages, Memorandum, University of California, Berkeley, 1973.
37. J. VAN LEEUWEN, A generalization of Parikh’s theorem in formal language theory, in “Automata, Languages, and Programming, 2nd Colloq.” (J. Loeckx, Ed.), *Lecture Notes in Computer Science* No. 14, pp. 17–26, Springer-Verlag, Berlin, 1974.
38. J. VAN LEEUWEN, Variations of a new machine model, in “Conf. Record 17th Ann. IEEE Sympos. on Foundations of Computer Science, Houston, Texas, 1976,” pp. 228–235.