LINEAR ITERATED PUSHDOWNS*

David J. Weir

School of Cognitive and Computing Sciences

University of Sussex

Brighton, BN1 9QH

(+44 273) 678294

*Thanks to Joost Engelfriet for bringing work on iterated pushdown to the author's attention. Thanks also to Aravind Joshi, Bernard Lang and K. Vijay-Shanker for helpful comments.

Abstract

This paper discusses variants of nondeterministic one-way S-automata and context-free S-grammars where S is a storage type. The framework that these systems provide can be used to give alternative formulations of embedded pushdown automata and linear indexed grammars. The embedded pushdown automata is obtained by means of a linear version of a class of storage types called iterated pushdowns. Linear indexed grammar is obtained by using the pushdown storage type and restricting the way in which the grammar uses its storage.

Key Words

Grammar formalisms, string automata, mathematical linguistics, formal language theory.

1 Introduction

This paper concerns the problem using string automata to describe the class of tree adjoining languages (tal). Providing an automaton-theoretic characterization of a grammar formalism leads to insights into the parsing problem for that formalism. For example, in the case of context-free grammars, Lang (1991) describes a framework that can be used to compare a variety of parsing algorithms in which the pushdown automata (pda) plays a central role. Automata are also used to characterize deterministic subclasses of grammar formalisms. Vijay-Shanker (1987) gives an automaton model for tal in the form of the embedded pushdown automata (epda) and a bottom-up version of this automaton was used to investigate deterministic tree adjoining languages (Schabes & Vijay-Shanker, 1990).

Justification, (given by Vijay-Shanker, 1987), for the attraction of the epda was that, apart from its use of an embedded pushdown, it is identical to the pda; and furthermore, that the embedded pushdown is a natural generalization of the pushdown. Additional support for this argument comes from the characterization of an infinite progression of automata models having pda and epda as its first two members (Weir, 1992). The goal of this paper is to back this up formally with support for the following two claims.

Claim 1: the pda and epda can be seen as different instantiations of the same general automata model.

Claim 2: the embedded pushdown can be obtained from the pushdown by use of an operation on forms of storage.

The close relationship between pda and epda is not unique. Many string automaton models share various characteristics: they are nondeterministic; they are one-way; and they include a finite state control. Just as in the case of epda and pda it is the nature of the additional storage that varies from machine to machine. In light of this, the notion of a nondeterministic one-way S-automaton for storage type S was defined (Engelfriet, 1982, 1983). This was intended to provide a general framework within which all such automata can be described (following Scott, 1967). A particular class of automata is obtained with suitable instantiation of the storage type S. For example, in the case where S is a standard pushdown we get automata that are a trivial variant of pda. We give support to Claim 1 above by identifying a storage type that gives rise to epda.

The embedded pushdown storage type is obtained by the iteration of operators on storage types (Greibach, 1970; Maslov, 1976; Engelfriet, 1982, 1983). Iteration of a pushdown operator gives rise to a progression of automata that characterize the OI-(string) language hierarchy (Damm & Goerdt, 1986). Vogler (1986) it is shown that iteration of a one-turn pushdown operator gives a characterization of the hierarchy described by Khabbaz (1974). We satisfy Claim 2 by giving a new operator that gives the embedded pushdown storage type from the pushdown storage type.

The paper is organized as follows. In order to make the paper self-contained, in Section 2 we present the original definition of the epda given by Vijay-Shanker (1987). In Section 3 we define storage types, S-automata and the pushdown operator $\mathcal{P}$ on storage types (Greibach, 1970; Engelfriet, 1982, 1983). Then, in Section 4, we present a new operator on storage types called the linear pushdown operator. When this operator is applied to a pushdown it gives a reformulation of the embedded pushdown of epda and when it is iterated it gives a reformulation of the hierarchy described by Weir (1992).

The notion of storage types is not only useful in defining classes of automata but has also been applied to grammars that use storage. In Section 5, we present the notion of a context-free S-grammar (Engelfriet, 1982; Engelfriet & Vogler, 1986) and show how linear indexed grammars (lig) (Gazdar, 1988) can be expressed within this framework by defining context-free linear S-grammars. Lig[1] are known to be weakly equivalent to tag (Vijay-Shanker & Weir, in press) and are very similar to epda.

2 Embedded Pushdown Automata

The definition of the epda given here is based on that given by Vijay-Shanker (1987). Given a set of pushdown symbols Γ, pushdowns are members of Γ^* and are denoted $\sigma_n \ldots \sigma_1$ where each σ_i is a pushdown symbol ($0 \leq i \leq n$) and σ_1 is the top pushdown symbol of this pushdown. The embedded pushdown that contains the $m \geq 1$ non-empty pushdowns $\sigma_{m,k_m} \ldots \sigma_{m,1}$, ..., $\sigma_{2,k_2} \ldots \sigma_{2,1}$ and $\sigma_{1,k_1} \ldots \sigma_{1,1}$, (with $\sigma_{1,k_1} \ldots \sigma_{1,1}$ on top) is denoted

$$\ddagger \sigma_{m,k_m} \ldots \sigma_{m,1} \ldots \ddagger \sigma_{2,k_2} \ldots \sigma_{2,1} \ddagger \sigma_{1,k_1} \ldots \sigma_{1,1}$$

[1] The name linear indexed grammars has been used by Duske and Parchmann (1984) to name a different restriction of indexed grammars in which only one nonterminal can appear on the right of production.

where $\ddagger$ is a special symbol not in Γ that marks the beginning of each embedded pushdown and each $k_i \geq 1$ $(1 \leq i \leq m)$. Thus, embedded pushdowns are members of $(\ddagger\Gamma^+)^*$. We use the symbol Υ (with or without subscripts) to denote members of $(\ddagger\Gamma^+)$, thus the above embedded pushdown can be denoted $\Upsilon_m \ldots \Upsilon_1$.

Definition 2.1 An epda is a seven tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_F, \sigma_0)$ where

Q is a finite set of states,

Σ is a finite set of input symbols,

Γ is a finite set of pushdown symbols,

$q_0 \in Q$ is the start state,

$Q_F \subseteq Q$ is the set of final states,

σ_0 is the initial pushdown symbol and

δ is the transition function.

The transition function δ is a mapping from $Q \times \Sigma_\lambda \times \Gamma$ to finite subsets of $Q \times (\ddagger\Gamma^+)^* \times \Gamma^* \times (\ddagger\Gamma^+)^*$ where λ is the empty string, $\Sigma_\lambda = \Sigma \cup \{ \lambda \}$ and $\ddagger \notin \Gamma$.

A configuration of M is a member of $Q \times (\ddagger\Gamma^+)^* \times \Sigma^* \times \Sigma^*$ where $\langle q, \Upsilon_m \ldots \Upsilon_1, w_1, w_2 \rangle$ indicates that M is in state q with embedded pushdown $\Upsilon_m \ldots \Upsilon_1$, having consumed the initial part w_1 of the input $w_1 w_2$.

The computation relation $\vdash_M$ is defined such that

$$\langle q, \Upsilon_m \ldots \Upsilon_2 \ddagger \sigma_{1,k} \ldots \sigma_{1,2}\sigma_{1,1}, w_1, \varepsilon w_2 \rangle$$
$$\vdash_M \langle q', \Upsilon_m \ldots \Upsilon_2 \Upsilon'_{m'} \ldots \Upsilon'_1 \ddagger \sigma_{1,k} \ldots \sigma_{1,2}\sigma_n \ldots \sigma_1 \Upsilon''_{m''} \ldots \Upsilon''_1, w_1\varepsilon, w_2 \rangle$$

if and only if

$$\langle q', \Upsilon'_{m'} \ldots \Upsilon_1, \sigma_n \ldots \sigma_1, \Upsilon''_{m''} \ldots \Upsilon''_1 \rangle \in \delta(q, \varepsilon, \sigma_{1,1})$$

Note that if $n = 0$ and $k = 1$ then the top of the old pushdown has been emptied and is no longer present in the new embedded pushdown.

The above component of δ indicates that M can make the following move: change state from q to q'; consume the (possibly empty) input symbol ε; insert m' new pushdowns $\Upsilon'_{m'} \ldots \Upsilon'_1$ below the top pushdown of the old embedded pushdown; insert m'' new pushdowns $\Upsilon''_{m''} \ldots \Upsilon''_1$ above the top pushdown of the old embedded pushdown; and modify the top pushdown of the old embedded pushdown by replacing its top symbol with the possibily empty string of new pushdown symbols $\sigma_n \ldots \sigma_1$.

The language accepted by the epda $M = (Q, \Sigma, \Gamma, \delta, q_0, Q_F, \sigma_0)$ by final state is defined as follows

$$L(M) = \{w \mid \langle q_0, \ddagger\sigma_0, \lambda, w\rangle \vdash_M^* \langle q_f, \Upsilon_m \ldots \Upsilon_1, w, \lambda\rangle \text{ for some } q_f \in Q_F \text{ and some embedded pushdown } \Upsilon_m \ldots \Upsilon_1 \text{ where } m \geq 0\}$$

3 Storage Types, S-Automata and Pushdowns of S

There are many ways of extending finite automata with the addition of some form of "unbounded" memory. What varies from model to model is the nature of this memory, in particular, the operations that can be performed on it in a computation step. Storage types are intended to formalize the different kinds of memory that automata can use, without reference to particular automaton models. The notation and definitions in this section are taken from Vogler (1986).

Definition 3.1 A storage type is a five tuple $S = (C, C_0, P, F, m)$ where

C is the set of configurations,

$C_0 \subseteq C$ is the set of initial configurations,

P is a set of predicates,

F is a set of instructions and

m is the meaning function such that

$m(p) : C \to \{\text{TRUE}, \text{FALSE}\}$ for each predicate $p \in P$ and

$m(f) : C \to C$ for each instruction $f \in F$ ($m(f)$ may be partial).

Predicates test configurations and are used in automata to determine if a move should be made. Instructions are functions from "old" configurations to "new" configurations, e.g., to push or pop symbols from a pushdown.

We now define S-automata for a storage type $S = (C, C_0, P, F, m)$.

Definition 3.2 A S-automaton is a six tuple $M = (Q, \Sigma, \delta, q_0, c_0, \tilde{Q})$ where

Q is a finite set of states,

Σ is the input alphabet,

$q_0 \in Q$ is the initial state,

$c_0 \in C_0$ is the initial configuration, $\tilde{Q} \subseteq Q$ is the set of final states and

δ is the transition function where

$\delta \subseteq_f Q \times \Sigma_\lambda \times \text{BE}(P) \times Q \times F$ where

$\subseteq_f$ denotes finite subset and

$\text{BE}(P)$ is the set of boolean expressions over P.

A global configuration of M is a member of $Q \times \Sigma^* \times C$ such that if M is in configuration (q, w, c) then M is in state q, the string w remains on input tape and the configuration of storage is c.

The computation relation $\vdash_M$ is defined such that

$$(q_1, xw, c_1) \vdash_M (q_2, w, c_2) \quad \text{iff} \quad (q_1, x, \pi, q_2, f) \in \delta, \; m(\pi)(c_1) = \text{TRUE} \quad \text{and} \quad m(f)(c_1) = c_2$$

The language accepted by M is

$$L(M) = \{w \in \Sigma^* \mid \; (q_0, w, c_0) \vdash_M^* (q_f, \lambda, c) \text{ for some } q_f \in \tilde{Q} \text{ and } c \in C\}$$

We now define $\mathcal{P}$, a pushdown operator that produces one storage type given another. Given a storage type $S = (C, C_0, P, F, m)$ the storage type $\mathcal{P}(S)$ is called the pushdown of S. A configuration of $\mathcal{P}(S)$ is a string $(\gamma_1, c_1) \ldots (\gamma_n, c_n)$ where $n > 0$ and for each $i \in \{1, \ldots, n\}$ γ_i is a pushdown symbol and $c_i \in C$ is a S-configuration. The predicates of $\mathcal{P}(S)$ test the top element (γ_1, c_1) either by testing c_1 with predicates in P or by checking the identity of γ_1. The instructions of $\mathcal{P}(S)$ produce new configurations of $\mathcal{P}(S)$ either by applying an instruction from F to c_1 or by pushing or poping new elements.

Given a storage type $S = (C, C_0, P, F, m)$ the pushdown of S is defined as follows.

Definition 3.3 The pushdown of S is a five tuple $\mathcal{P}(S) = (C', C_0', P', F', m')$ where

$C' = (\Gamma \times C)^+$ where Γ is a fixed, infinite set of pushdown symbols,

$C_0' = \{ (\gamma, c_0) \mid \gamma \in \Gamma \text{ and } c_0 \in C_0 \}$,

$P' = \{ \text{TOP} = \gamma \mid \gamma \in \Gamma \} \cup \{ \text{TEST}(p) \mid p \in P \}$,

$F' = \{ \text{PUSH}(\gamma, f) \mid \gamma \in \Gamma \text{ and } f \in F \} \cup \{ \text{POP} \} \cup \{ \text{STAY}(\gamma) \mid \gamma \in \Gamma \} \cup \{ \text{STAY} \}$ and

for each $\sigma, \gamma \in \Gamma$, $c \in C$, $\beta \in (\Gamma \times C)^*$, $p \in P$ and $f \in F$

$m'(\text{TOP} = \gamma)((\sigma, c)\beta) = (\sigma = \gamma)$,

$m'(\text{TEST}(p))((\sigma, c)\beta) = m(p)(c)$,

$m'(\text{PUSH}(\gamma, f))((\sigma, c)\beta) = (\gamma, m(f)(c))(\sigma, c)\beta$,

$m'(\text{POP})((\sigma, c)\beta) = \beta$ if $\beta \neq \lambda$ and undefined otherwise,

$m'(\text{STAY}(\gamma))((\sigma, c)\beta) = (\gamma, c)\beta$ and

$m'(\text{STAY})((\sigma, c)\beta) = (\sigma, c)\beta$.

The pushdown operator $\mathcal{P}$ is iterated by letting $\mathcal{P}^0(S) = S$ and for every $k \geq 0$, $\mathcal{P}^{k+1}(S) = \mathcal{P}(\mathcal{P}^k(S))$. An infinite progression of storage types is defined as follows. First define the trivial storage type: $S_0 = (\{c\}, \{c\}, \emptyset, \{\text{id}\}, m)$ where c is arbitrary object and $m(\text{id})(c) = c$. The kth storage type in the progression, denoted $\mathcal{P}^k$, is defined as $\mathcal{P}^k = \mathcal{P}^k(S_0)$.

It is known that $\mathcal{P}^k$-automaton give a strict hierarchy, that $\mathcal{P}^0$-automaton accept the class of regular languages, that $\mathcal{P}^1$-automaton accept the class of context-free languages and that the progression characterizes the OI-(string) language hierarchy (Damm & Goerdt, 1986). $\mathcal{P}^2$-automaton are equivalent to indexed pushdown-automata of Parchmann, Duske & Specht (1980) which accept the class of indexed languages (Aho, 1968).

Example 3.1 Consider the storage type $\mathcal{P}^2 = \mathcal{P}(\mathcal{P}(S_0)) = (C'', C_0'', P'', F'', m'')$ where $\mathcal{P}(S_0) = (C', C_0', P', F', m')$ and $S_0 = (\{c\}, \{c\}, \emptyset, \{\text{id}\}, m)$.

An example configuration in C'' is:

$$(\gamma_1, (\gamma_2, c)(\gamma_3, c))(\gamma_4, (\gamma_5, c))$$

Applying the predicates $\text{TOP} = \gamma$ and $\text{TEST}(\text{TOP} = \gamma)$ from P'' to this configuration we get:

$$\begin{aligned} &m''(\text{TOP} = \gamma)((\gamma_1, (\gamma_2, c)(\gamma_3, c))(\gamma_4, (\gamma_5, c))) \\ &= (\gamma = \gamma_1) \end{aligned}$$

$$\begin{aligned} &m''(\text{TEST}(\text{TOP} = \gamma))((\gamma_1, (\gamma_2, c)(\gamma_3, c))(\gamma_4, (\gamma_5, c))) \\ &= m'(\text{TOP} = \gamma)((\gamma_2, c)(\gamma_3, c)) \\ &= (\gamma = \gamma_2) \end{aligned}$$

Applying the instructions $\text{PUSH}(\gamma, \text{PUSH}(\gamma', \text{id}))$ and POP from F'' to this configuration

we get:

$$m''(\text{PUSH}(\gamma, \text{PUSH}(\gamma', \text{id})))\,((\gamma_1,(\gamma_2,c)(\gamma_3,c))(\gamma_4,(\gamma_5,c)))$$
$$= (\gamma, m'(\text{PUSH}(\gamma', \text{id}))\,((\gamma_2,c)(\gamma_3,c)))(\gamma_1,(\gamma_2,c)(\gamma_3,c))(\gamma_4,(\gamma_5,c))$$
$$= (\gamma,(\gamma',c)(\gamma_2,c)(\gamma_3,c))(\gamma_1,(\gamma_2,c)(\gamma_3,c))(\gamma_4,(\gamma_5,c))$$

$$m''(\text{POP})\,((\gamma_1,(\gamma_2,c)(\gamma_3,c))(\gamma_4,(\gamma_5,c)))$$
$$= (\gamma_4,(\gamma_5,c))$$

4 Linear Iteration Of Pushdowns

Since $\mathcal{P}^2$-automata accept indexed languages and are, therefore, more powerful than epda our goal is to find a new operator that gives the storage type of the epda. We will define a variant of the operator $\mathcal{P}$ called $\mathcal{P}_{\text{lin}}$ which we call the linear pushdown of S. $\mathcal{P}_{\text{lin}}$ will be defined in such a way that $\mathcal{P}^0_{\text{lin}}$-automaton accept the regular languages, $\mathcal{P}^1_{\text{lin}}$-automaton accept the context-free languages and $\mathcal{P}^2_{\text{lin}}$-automaton accept the tal.

In order to understand why P_2-automata are more powerful than epda consider the push instruction of the storage type $\mathcal{P}^2$:

$$m'(\text{PUSH}(\gamma, f))\,((\sigma,c)\beta) = (\gamma, m(f)(c))(\sigma,c)\beta$$

After the push move is complete a copy of the pushdown (σ,c) is left behind just below the new top pushdown $(\gamma, m(f)(c))$. It is the possibility of copying unbounded configurations (in this case c) that must be ruled out in the instructions of $\mathcal{P}_{\text{lin}}$. For example, we do not wish to allow the following: $m'(\text{PUSH}(\gamma, \text{STAY}))\,((\sigma,c)\beta) = (\gamma,c)(\sigma,c)\beta$.

Before defining the new operator $\mathcal{P}_{\text{lin}}$ we need to redefine storage types to have a single initial configuration rather than a set. Thus, a storage type is denoted (C, c_0, P, F, m) where C, P, F, and m are as before and $c_0 \in C$ is the initial configuration. Likewise, we redefine the trivial storage type $S_0 = (\{\,c\,\}, c, \emptyset, \{\,\text{id}\,\}, m)$. Note that having made this change there is no need to include the fifth component, c_0, of an S-automaton giving the initial configuration of the machine. Thus, an S-automaton is defined as a five tuple $M = (Q, \Sigma, \delta, q_0, \tilde{Q})$.

In order to define initial configurations for storage types produced by iterating $\mathcal{P}_{\text{lin}}$ we designate γ_0 to be a distinguished member of Γ (the set of pushdown symbols) that is the initial pushdown symbol.

Given a storage type $S = (C, c_0, P, F, m)$ the linear pushdown of S is defined as follows.

Definition 4.1 The linear pushdown of S is a five tuple $\mathcal{P}_{\text{lin}}(S) = (C', c_0', P', F', m')$ where

$C' = (\Gamma \times C)^+$ where Γ is a fixed, infinite set of pushdown symbols,

$c_0' = (\gamma_0, c_0)$ is the initial configuration,

$P' = \{ \text{TOP} = \gamma \mid \gamma \in \Gamma \} \cup \{ \text{TEST}(p) \mid p \in P \}$,

$F' = \{ \text{PUSH}(\alpha, f, i) \mid \alpha \in \Gamma^+,\ f \in F \text{ and } 1 \leq i \leq |\alpha| \} \cup \{ \text{POP} \} \cup \{ \text{STAY} \}$ and

for each $\sigma, \gamma \in \Gamma$, $\alpha \in \Gamma^+$, $1 \leq i \leq |\alpha|$, $c \in C$, $\beta \in (\Gamma \times C)^*$, $p \in P$ and $f \in F$

$m'(\text{TOP} = \gamma)((\sigma, c)\beta) = (\sigma = \gamma)$,

$m'(\text{TEST}(p))((\sigma, c)\beta) = m(p)(c)$,

$m'(\text{PUSH}(\gamma_1 \ldots \gamma_i \ldots \gamma_n, f, i))((\sigma, c)\beta)$

$= (\gamma_1, c_0) \ldots (\gamma_{i-1}, c_0)(\gamma_i, m(f)(c))(\gamma_{i+1}, c_0) \ldots (\gamma_n, c_0)\beta$,

$m'(\text{POP})((\sigma, c)\beta) = \beta$ if $\beta \neq \lambda$ and undefined otherwise,

$m'(\text{STAY})((\sigma, c)\beta) = (\sigma, c)\beta$.

In other words, the push move $\text{PUSH}(\alpha, f, i)$ involves *replacing* the top subpushdown with $|\alpha|$ new subpushdowns only the ith of which contains the result of applying f, the others containing the initial pushdown of S. Note that the definition of the other instructions and of the predicates are unchanged from the earlier definition of $\mathcal{P}$. We ommitted the instruction $\text{STAY}(\gamma)$ since $\text{STAY}(\gamma)$ corresponds to $\text{PUSH}(\gamma, \text{STAY}, 1)$.

The linear pushdown operator can be iterated in the same way as the pushdown operator, i.e., $\mathcal{P}_{\text{lin}}^k = \mathcal{P}_{\text{lin}}^k(S_0)$.

Example 4.1 Consider $\mathcal{P}_{\text{lin}}^2 = P_{\text{lin}}(P_{\text{lin}}(S_0)) = (C'', c_0'', P'', F'', m'')$ where $\mathcal{P}_{\text{lin}}(S_0) = (C', c_0', P', F', m')$ and $S_0 = (\{ c \}, c, \emptyset, \{ \text{id} \}, m)$.

The initial configuration $c_0'' = (\gamma_0, (\gamma_0, c))$

An example configuration in C'' is:

$$(\gamma_1, (\gamma_2, c))(\gamma_4, (\gamma_5, c))$$

Applying the push move $\text{PUSH}(\gamma\gamma'\gamma'', \text{PUSH}(\gamma''', \text{id}, 1), 2)$ of P'' to this configuration we

get:

$$
\begin{aligned}
& m''(\textsc{push}(\gamma\gamma'\gamma'', \textsc{push}(\gamma''', \mathrm{id}, 1), 2))\,((\gamma_1, (\gamma_2, c))(\gamma_4, (\gamma_5, c))) \\
&= (\gamma, c_0)(\gamma', m'(\textsc{push}(\gamma''', \mathrm{id}, 1))\,((\gamma_2, c)))(\gamma'', c_0)(\gamma_4, (\gamma_5, c)) \\
&= (\gamma, c_0)(\gamma', (\gamma''', c))(\gamma'', c_0)(\gamma_4, (\gamma_5, c))
\end{aligned}
$$

$\mathcal{P}^0$-automata and $\mathcal{P}^0_{\mathrm{lin}}$-automata both recognize the regular languages. Since the storage types $\mathcal{P}(S_0)$ and $\mathcal{P}_{\mathrm{lin}}(S_0)$ are equivalent, $\mathcal{P}^1$-automata and $\mathcal{P}^1_{\mathrm{lin}}$-automata correspond to the pda. It is also clear that $\mathcal{P}^2_{\mathrm{lin}}$-automata is a reformulation of epda and that for each $k > 1$ $\mathcal{P}^k_{\mathrm{lin}}$-automata is a reformulation of the kth automaton class in the hierarchy defined by Weir (1992).

5 Context-Free S-Grammars

The notion of storage types has also been applied to grammars (Engelfriet, 1982; Engelfriet & Vogler, 1986) to give, for example, the context-free S-grammars in which configurations of S are associated with nonterminals of the grammar. If S is a pushdown then we get a grammar formalism that is equivalent to indexed grammars (Engelfriet, 1982; Engelfriet & Vogler, 1986). We consider a restriction on the use of the storage type S such that only one nonterminal on the right of productions inherits the storage associated with the nonterminal on the left. In this case we find that instantiating S to be standard pushdowns gives us lig.

Let $S = (C, c_0, P, F, m)$ be a storage type[2]

Definition 5.1 A context-free S-grammars is a four tuple $G = (V_N, V_T, R, A_{in})$ where

V_N finite set of nonterminal symbols,

V_T finite set of terminal symbols,

$A_{in} \in V_N$ is the start symbol and

R finite set of rules of the form $A \to \textsc{if}\ \pi\ \textsc{then}\ \alpha$ where

$A \in V_N$,

$\pi \in \textsc{be}(P)$,

$\alpha \in (V_N[\,F\,] \cup V_T)^*$ and

$V_N[\,F\,] = \{\, A[\,f\,] \mid A \in V_N \text{ and } f \in F \,\}$.

[2]Note that for simplicity we continue to use the revised definition of a storage type with a single initial configuration.

The set of sentential forms of $G = (V_N, V_T, R, A_{in})$ is a subset of

$$(V_N[\,C\,] \cup V_T)^* \text{ where } V_N[\,C\,] = \{\, A[\,c\,] \mid A \in V_N \text{ and } c \in C \,\}$$

The derives relation $\underset{G}{\Longrightarrow}$ is defined such that

$$\begin{aligned}
\varphi_1 \underset{G}{\Longrightarrow} \varphi_2 \quad \text{iff} \quad & \varphi_1 = u_1 A[\,c\,] u_2 \text{ for some } u_1, u_2 \in (V_N[\,C\,] \cup V_T)^*, \\
& A \rightarrow \text{IF } \pi \text{ THEN } \alpha \in R, \\
& c \in C \text{ such that } m(\pi)(c) = \text{TRUE}, \\
& \varphi_2 = u_1 \varphi u_2 \text{ and} \\
& \varphi \text{ is produced from } \alpha \text{ by replacing each } B[\,f\,] \text{ by } B[\,m(f)(c)\,]
\end{aligned}$$

$$L(G) = \left\{ w \in V_T^* \;\middle|\; A_{in}[\,c_0\,] \underset{G}{\overset{*}{\Longrightarrow}} w \right\}$$

Note that context-free S_0-grammars are equivalent to context-free grammars and that context-free $\mathcal{P}(S_0)$-grammars are equivalent to indexed grammars (Engelfriet, 1982; Engelfriet & Vogler, 1986). Thus, since the storage types $\mathcal{P}(S_0)$ and $\mathcal{P}_{\text{lin}}(S_0)$ are equivalent $\mathcal{P}_{\text{lin}}(S_0)$-grammars also correspond to indexed grammars. In order to obtain a formalism that corresponds to lig we need to modify the definition of S-grammars. The modifications made to context-free S-grammars mirror the changes made to $\mathcal{P}$ to get $\mathcal{P}_{\text{lin}}$.

Consider storage type (C, c_0, P, F, m).

Definition 5.2 A context-free linear S-grammar is a four tuple $G = (V_N, V_T, R, A_{in})$

where

V_N finite set of nonterminal symbols,

V_T finite set of terminal symbols,

$A_{in} \in V_N$ is the start symbol and

R finite set of rules of the form $A \rightarrow \text{IF } \pi \text{ THEN } \alpha$ where

$A \in V_N$,

$\pi \in \text{BE}(P)$ and

$\alpha \in \bar{V}^* V_N[\,F\,] \bar{V}^* \cup \bar{V}^*$ where $\bar{V} = V_T \cup V_N[\,]$.

In other words the right-hand-side of productions have at most one occurence of an instruction f in the context: $B[\,f\,]$. All other nonterminals appear in the context $B[\,]$. Suppose that the S-configuration c is associated with the nonterminal A on the left-hand-side of the production.

Application of the production can associate the S-configuration $m(f)(c)$ for some $f \in F$ with at most one of the nonterminals on the right-hand-side. All other nonterminals will be associated with the initial configuration c_0. Thus, the derives relation $\underset{G}{\Longrightarrow}$ is defined such that

$$\begin{aligned}\varphi_1 \underset{G}{\Longrightarrow} \varphi_2 \quad \text{iff} \quad & \varphi_1 = u_1 A[\,c\,]u_2 \text{ for some } u_1, u_2 \in (V_N[\,C\,] \cup V_T)^* \\ & A \rightarrow \text{IF } \pi \text{ THEN } \alpha \in R, \\ & c \in C \text{ such that } m(\pi)(c) = \text{TRUE}, \\ & \varphi_2 = u_1 \varphi u_2, \\ & \varphi \text{ is produced from } \alpha \text{ by replacing } B[\,f\,] \text{ by } B[\,m(f)(c)\,] \text{ and} \\ & \qquad \text{by replacing each } B[\,] \text{ by } B[\,c_0\,]\end{aligned}$$

$$L(G) = \left\{ w \in V_T^* \;\middle|\; A_{in}[\,c_0\,] \underset{G}{\overset{*}{\Longrightarrow}} w \right\}$$

Example 5.1 The $\mathcal{P}^1_{\text{lin}}$-grammar $G = (\{\, A_{in}, B \,\}, \{\, a, b, c, d \,\}, R, A_{in})$ generates the language $\{\, a^n b^n c^n d^n \mid n \geq 0 \,\}$ where R contains the productions:

$$\begin{aligned}A_{in} &\rightarrow \text{IF TOP} = \gamma_0 \text{ THEN } aA_{in}[\,\text{PUSH}(\gamma_1\gamma_0, id, 1)]d \\ A_{in} &\rightarrow \text{IF TOP} = \gamma_1 \text{ THEN } aA_{in}[\,\text{PUSH}(\gamma_1\gamma_1, id, 1)]d \\ A_{in} &\rightarrow \text{IF TOP} = \gamma_0 \text{ OR TOP} = \gamma_1 \text{ THEN } B[\,\text{STAY}\,] \\ B &\rightarrow \text{IF TOP} = \gamma_1 \text{ THEN } bB[\,\text{POP}\,]c \\ B &\rightarrow \text{IF TOP} = \gamma_0 \text{ THEN } \lambda\end{aligned}$$

6 Conclusions

We have used the notion of storage types and operators on storage types to reformulate two systems that characterize the tree adjoining languages: embedded pushdown automata and linear indexed grammars. Not only does this help to clarify the relationship between tree adjoining languages and context-free languages, but the use of storage types has additional benefits.

Defining grammars and automata in terms of the storage type that they use has proved to be a useful way of relating the power of different systems (e.g. see Engelfriet & Vogler, 1986). This technique involves the notions of storage type simulation and storage type equivalence. Two storage types are equivalent if they can simulate each other. A storage type $S = (C, C_0, P, F, m)$ can simulate a storage type $S' = (C', C'_0, P', F', m')$ if there is a partial function from C to C'

that specifies which configurations of C simulate configurations of C'. In addition, S must be able to simulate each member of P' and F' (this can be specified with a deterministic flowchart using members of P and F).

General theorems can be established that make use of storage type equivalence to show that different systems characterize the same class of languages (e.g. see Engelfriet & Vogler, 1986, Theorem 4.18). In a similar way, it is possible to show that since $\mathcal{P}^k$ can simulate $\mathcal{P}^k_{\text{lin}}$, $\mathcal{P}^k$-automata are more powerful than $\mathcal{P}^k_{\text{lin}}$-automata.

Reference

Aho, A. V. (1968). Indexed grammars — An extension to context free grammars. *J. ACM*, *15*, 647–671.

Damm, W., & Goerdt, A. (1986). An automata-theoretic characterization of the OI-hierarchy. *Inf. Control*, *71*, 1–32.

Duske, J., & Parchmann, R. (1984). Linear indexed languages. *Theor. Comput. Sci.*, *32*, 47–60.

Engelfriet, J. (1982). Recursive automata. Unpublished notes.

Engelfriet, J. (1983). Iterated pushdown automata and complexity classes. In *Proc. 15th STOC*, pp. 365–373.

Engelfriet, J., & Vogler, H. (1986). Pushdown machines for the macro tree transducer. *Theor. Comput. Sci.*, *42*, 251–368.

Gazdar, G. (1988). Applicability of indexed grammars to natural languages. In Reyle, U., & Rohrer, C. (Eds.), *Natural Language Parsing and Linguistic Theories*, pp. 69—94. D. Reidel, Dordrecht, Holland.

Greibach, S. A. (1970). Full AFL's and nested iterative substitution. *Inf. Control*, *16*, 7–35.

Khabbaz, N. A. (1974). A geometric hierarchy of languages. *J. Comput. Syst. Sci.*, *8*, 142–157.

Lang, B. (1991). A uniform framework for parsing. In Tomita, M. (Ed.), *Current Issues in Parsing Technology*, pp. 153–171. Klumer Academic Publisher.

Maslov, A. N. (1976). Multilevel stack automata. *Problemy Peredachi Informatsii*, *12*, 55–62.

Parchmann, R., Duske, J., & Specht, J. (1980). On the determinsitic indexed languages. *Inf. Control*, *45*, 48—67.

Schabes, Y., & Vijay-Shanker, K. (1990). Deterministic left to right parsing of tree adjoining grammars. In 28^{th} *meeting Assoc. Comput. Ling.*, pp. 276—283.

Scott, D. (1967). Some definitional suggestions for automata theory. *J. Comput. Syst. Sci.*, *4*, 187–212.

Vijay-Shanker, K. (1987). *A Study of Tree Adjoining Grammars.* Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Vijay-Shanker, K., & Weir, D. J. (in press). The equivalence of four extensions of context-free grammars. *Math. Syst. Theory.*

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

Weir, D. J. (1992). A geometric hierarchy beyond context-free languages. *Theor. Comput. Sci.*, *104*, 235–261.