$LR(0)$ Conjunctive Grammars and Deterministic Synchronized Alternating Pushdown Automata

Tamar Aizikowitz and Michael Kaminski

Department of Computer Science,
Technion – Israel Institute of Technology,
Haifa 32000, Israel

Abstract. In this paper we introduce a sub-family of synchronized alternating pushdown automata, *Deterministic Synchronized Alternating Pushdown Automata*, and a sub-family of conjunctive grammars, $LR(0)$ *Conjunctive Grammars*. We prove that deterministic SAPDA and $LR(0)$ conjunctive grammars have the same recognition/generation power, analogously to the classical equivalence between acceptance by empty stack of deterministic PDA and $LR(0)$ grammars. These models form the theoretical basis for efficient, *linear*, parsing of a rich sub-family of conjunctive languages, which *properly* includes all the boolean combinations of context-free $LR(0)$ languages.

1 Introduction

Context-free languages lay at the very foundations of Computer Science, proving to be one of the most appealing language classes for practical applications. On the one hand, they are quite expressive, covering such syntactic constructs as necessary, e.g., for mathematical expressions. On the other hand, they are polynomially parsable, making them practical for real world applications. However, research in certain fields, e.g., Computational Linguistics [4,8], has raised a need for computational models which extend context-free languages.

Conjunctive Grammars (CG) are an example of such a model. Introduced by Okhotin in [9], CG are a generalization of context-free grammars which allow explicit conjunction operations in rules, thereby adding the power of intersection. CG were shown by Okhotin to accept all finite intersections of context-free languages, as well as some additional languages. Okhotin proved the languages generated by these grammars to be polynomially parsable [9,12,13], making the model practical from a computational standpoint, and therefore, of interest for applications in various fields such as, e.g., programming languages.

Alternating automata models were first introduced by Chandra, Kozen and Stockmeyer in [3]. Alternating Pushdown Automata were further explored in [7], and shown to accept exactly the exponential time languages. As such, they are too strong a model for Conjunctive Grammars. Synchronized Alternating

A. Kulikov and N. Vereshchagin (Eds.): CSR 2011, LNCS 6651, pp. 345–358, 2011.

Pushdown Automata (SAPDA), introduced in [1], are a weakened version of Alternating Pushdown Automata, which, in particular, accept intersections of context-free languages. In [1], SAPDA were proven to be equivalent[1] to CG.

Deterministic context-free languages are a sub-family of context-free languages which can be accepted by a deterministic PDA. In [6], Knuth introduced the notion of $LR(k)$ grammars, and proved their equivalence to deterministic PDA. Through this equivalence, he developed a linear time LR parsing algorithm for deterministic context-free languages, which quickly became the basis of modern-day compilation theory. Furthermore, Knuth proved that $LR(0)$ languages (those which can be parsed with no lookahead) are equivalent to deterministic PDA accepting by empty stack.

In [11], Okhotin presented an extension of Tomita's Generalized LR parsing algorithm [14] for CG. The algorithm utilizes non-deterministic LR parsing, and works for all conjunctive grammars in polynomial time. When applied deterministic context-free languages, the run-time is linear.

In this paper we introduce a sub-family of SAPDA, *Deterministic SAPDA* (DSAPDA), and a sub-family of CG, $LR(0)$ *Conjunctive Grammars*. We prove these sub-families are equivalent, analogously to the context-free case. Furthermore, we present a sophisticated and efficient implementation of DSAPDA, which forms the basis of a deterministic linear time parsing algorithm for $LR(0)$ conjunctive languages. This class of languages *properly* contains the context-free $LR(0)$ languages, thus expanding upon previous results.

2 Preliminaries

In this section, we recall the definitions of CG from [9], and SAPDA, from [1].

2.1 Conjunctive Grammars

Definition 1. *A* Conjunctive Grammar *is a tuple $G = (V, \Sigma, P, S)$, where V, Σ are disjoint finite sets of non-terminal and terminal symbols, $S \in V$ is the designated start symbol, and P is a finite set of rules of the form $A \to (\alpha_1 \,\&\, \cdots \,\&\, \alpha_n)$ s.t. $A \in V$ and $\alpha_i \in (V \cup \Sigma)^*$, $i = 1, \ldots, n$. If $n = 1$, we just write $A \to \alpha_1$.*

Definition 2. Conjunctive formulas *over $V \cup \Sigma \cup \{(,), \&\}$ are defined as follows.*

- *All symbols of $V \cup \Sigma$, as well as ϵ, are conjunctive formulas.*
- *If $\mathcal{A}$ and $\mathcal{B}$ are formulas, then $\mathcal{AB}$ is a conjunctive formula.*
- *If $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are formulas, then $(\mathcal{A}_1 \,\&\, \cdots \,\&\, \mathcal{A}_n)$ is a conjunctive formula. We call each $\mathcal{A}_i$, $i = 1, \ldots, n$, a* conjunct *of $\mathcal{A}$.*[2]

Definition 3. *For a CG G, the relation of* immediate derivability*, $\Rightarrow_G$, on the set of conjunctive formulas is defined as follows.*

[1] We call two models equivalent if they accept/generate the same class of languages.
[2] Note that this definition is different from Okhotin's definition in [9].

1. $s_1\ A\ s_2 \Rightarrow_G s_1(\alpha_1\ \&\ \cdots\ \&\ \alpha_n)s_2$, *for* $A \to (\alpha_1\ \&\ \cdots\ \&\ \alpha_n) \in P$, *and*
2. $s_1\ (w\&\ \cdots\ \&w)\ s_2 \Rightarrow_G s_1\ w\ s_2$, *for* $w \in \Sigma^*$,

where $s_1, s_2 \in (V \cup \Sigma \cup \{(,),\&\})^*$. *We refer to 1 and 2 as* production *and* contraction *rules, respectively. As usual,* $\Rightarrow_G^*$ *is the reflexive and transitive closure of* $\Rightarrow_G$, *and the language of* G *is* $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

Example 1. ([9, Example 1]) The following CG G generates the non-context-free *multiple agreement* language $\{a^n b^n c^n | n \geq 0\}$. $G = (V, \Sigma, P, S)$, where $V = \{S, A, B, C, D\}$, $\Sigma = \{a, b, c\}$, and P consists of the following rules:

$$S \to (A\ \&\ C)\ ;\ A \to aA \mid B\ ;\ C \to Cc \mid D\ ;\ B \to bBc \mid \epsilon\ ;\ D \to aDb \mid \epsilon$$

$L(A) = \{a^m b^n c^n | m, n \geq 0\}$, and $L(C) = \{a^m b^m c^n | m, n \geq 0\}$. Therefore, $L(G) = L(A) \cap L(C) = \{a^n b^n c^n | n \geq 0\}$. For example, abc can be derived by

$$\begin{aligned} S &\Rightarrow (A\ \&\ C) \Rightarrow (aA\ \&\ C) \Rightarrow (aB\ \&\ C) \Rightarrow (abBc\ \&\ C) \Rightarrow (abc\ \&\ C) \\ &\Rightarrow (abc\ \&\ Cc) \Rightarrow (abc\ \&\ Dc) \Rightarrow (abc\ \&\ aDbc) \Rightarrow (abc\ \&\ abc) \Rightarrow abc\ . \end{aligned}$$

2.2 Synchronized Alternating Pushdown Automata

Introduced in [1], SAPDA extend standard PDA by adding the power of intersection. In an SAPDA, transitions are made to a conjunction of states. The model is non-deterministic, i.e., several conjunctions may be possible from a given configuration. If all conjunctions are of one state, the automaton is a standard PDA.

The stack memory of an SAPDA is a tree. Each leaf has a processing head which reads the input and writes to its branch independently. When a conjunctive transition is applied, the stack branch splits into multiple branches, one for each conjunct. When sibling branches empty, they must empty synchronously, i.e. after reading the same portion of the input, and in the same state, after which the computation continues from the parent branch.

Definition 4. *A* synchronized alternating pushdown automaton *is a tuple* $A = (Q, \Sigma, \Gamma, \delta, q_0, \bot)$, *where* δ *is a function that assigns to each element of* $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ *a finite subset of*

$$\{(q_1, \alpha_1) \wedge \cdots \wedge (q_k, \alpha_n) \mid n = 1, 2, \ldots,\ q_i \in Q \text{ and } \alpha_i \in \Gamma^*,\ i = 1, \ldots, n\}\ .$$

Everything else is as in the standard PDA model. Namely, Q *is a finite set of states,* Σ *and* Γ *are the input and the stack alphabets, respectively,* $q_0 \in Q$ *is the initial state, and* $\bot \in \Gamma$ *is the initial stack symbol, see, e.g., [5, pp. 107–112].*

Definition 5. *Let* A *be an SAPDA and let* $w \in \Sigma^*$.

- *The computation of* A *on* w *begins in state* q_0 *and with* $\bot$ *in the stack.*
- *Each step, a transition is applied to one of the non-empty branches.*
- *Sibling branches that are empty, in state* q, *and with the same remaining input, are collapsed, and the computation continues from the parent branch in state* q.

– *An accepting computation is one where the entire input is read, and the stack is emptied (i.e., all branches are emptied and collapsed).*

$L(A)$ is the language of all $w \in \Sigma^*$ s.t. A has an accepting computation on w.[3]

Example 2. [2, Example 6.2, pp. 64–65] We construct an SAPDA which accepts the language

$$L_{inf} = \{a^{i_1}b\ a^{i_2}b^2\ \cdots\ a^{i_n}b^n\ \$\ ba^{i_1}\ ba^{i_2}\ \cdots\ ba^{i_n}\ \$ \mid n \geq 1 \text{ and } i_1, \ldots, i_n \geq 1\}\ .$$

For this, we construct two automata, each in charge of a specific aspect of the language. The first, A_1, checks that the series of bs before the first \$ sign starts at 1 and increases by 1 at each step. The second, A_2, checks that the numbers of as before and after the first \$ match up appropriately. We then define an SAPDA A such that A accepts the intersection of the languages of A_1 and A_2. Following, we present the full construction of A_2. For the construction of A_1 and A, see [2, Example 6.2, pp. 64–65]. The SAPDA $A_2 = (Q_2, \Sigma, \Gamma_2, q_0, \bot, \delta_2)$ is defined as follows. $Q_2 = \{q_0, q_0', q_1, q_1', q_2, q_3, q_e\}$, $\Gamma_2 = \{\bot, a, b\}$, and

(1)	$\delta_2(q_0, a, \bot) = (q_1, a\bot) \wedge (q_0', \bot)$	(2)	$\delta_2(q_0', a, \bot) = (q_0', \bot)$
(3)	$\delta_2(q_0', b, \bot) = (q_0, \bot)$	(4)	$\delta_2(q_0, b, \bot) = (q_0, \bot)$
(5)	$\delta_2(q_0, \$, \bot) = (q_e, \bot)$	(6)	$\delta_2(q_1, a, \bot) = (q_1, a\bot)$
(7)	$\delta_2(q_1, a, a) = (q_1, aa)$	(8)	$\delta_2(q_1, b, a) = (q_1', ba)$
(9)	$\delta_2(q_1', b, b) = (q_1', bb)$	(10)	$\delta_2(q_1', a, b) = (q_2, b)$
(11)	$\delta_2(q_2, a, b) = (q_2, b)$	(12)	$\delta_2(q_2, b, b) = (q_2, b)$
(13)	$\delta_2(q_2, \$, b) = (q_3, b)$	(14)	$\delta_2(q_1', \$, b) = (q_3, b)$
(15)	$\delta_2(q_3, b, b) = (q_3, \epsilon)$	(16)	$\delta_2(q_3, a, b) = (q_3, b)$
(17)	$\delta_2(q_3, a, a) = (q_3', \epsilon)$	(18)	$\delta_2(q_3', a, a) = (q_3', \epsilon)$
(19)	$\delta_2(q_3', b, \bot) = (q_e, \bot)$	(20)	$\delta_2(q_3', \$, \bot) = (q_e, \epsilon)$
(21)	$\delta_2(q_e, a, \bot) = \delta_2(q_e, b, \bot) = (q_e, \bot)$	(22)	$\delta_2(q_e, \$, \bot) = (q_e, \epsilon)$

The automaton recursively opens a new branch for every first a in a series a^{i_j} that it sees. These branches subsequently store $a^{i_j}b^j$ in their stacks, and wait for the \$ sign. After the \$ is read, each branch "counts" to the jth series of a's by popping one b for each b encountered in the input. Thus, a^{i_j} will appear at the top of the stack after j b's have been read. If all a^{i_j} series before and after the \$ match, upon reading the final \$ sign, all branches will be able to empty their stacks, and collapse back to the root. See Figure 1 for sample configurations.

3 Deterministic SAPDA Model Definition

We define the notion of a deterministic SAPDA analogously to the classical definition of a deterministic PDA.

Definition 6. *An SAPDA $A = (Q, \Sigma, \Gamma, q_0, \delta, \bot)$ is* deterministic *if*

[3] For a detailed definition of SAPDA, see [2].

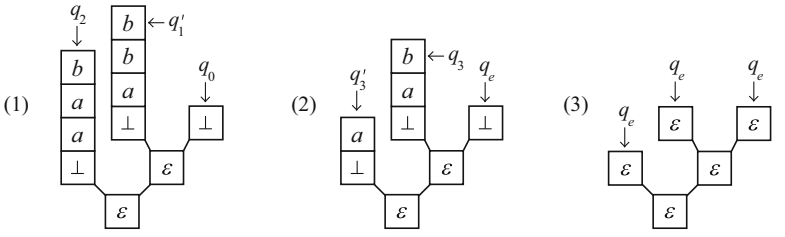

Fig. 1. Configurations of the automaton A_2, (1) after reading *aababb*, (2) after reading *aababb*\$*ba*, and (3) after reading *aababb*\$*baaba*\$

- *If* $\delta(q, \sigma, X) \neq \emptyset$ *for some* $\sigma \in \Sigma$, *then* $\delta(q, \epsilon, X) = \emptyset$.
- *For all* $q \in Q, \sigma \in \Sigma \cup \{\epsilon\}, X \in \Gamma, \quad |\delta(q, \sigma, X)| \leq 1$.

By Definition 6, a deterministic SAPDA has at most one computation on any given input word.[4] Note that the automaton A_2 from Example 2 is in fact a deterministic automaton, as is the full automaton for A for L_{inf}, see [2, Example 6.2, pp. 64–65].

3.1 Linear Membership for DSAPDA

We show that the membership problem for DSAPDA is decidable in linear time.

Remark 1. For the purposes of our discussion, we assume the automaton does not have an infinite series of ϵ transitions. As in the classical case, this assumption is not limiting (see [5, Lemma 10.3, p 236]), as ϵ-loops can be detected.

To proceed, we shall need the following notation. Let $A = (Q, \Sigma, \Gamma, q_0, \perp, \delta)$ be a DSAPDA. We denote by N_A the maximal number of branches opened in a single transition, and we denote by M_A the total number of different configurations possible for a branch head, i.e., $M_A = |Q| \times |\Gamma \cup \{\epsilon\}|$.

We consider an implementation model where the computation proceeds in rounds such that in each round, every branch takes one step. Note that this model is equivalent to the one where branches take steps in arbitrary order.

Consider a single stack-branch. As it behaves exactly like a standard deterministic PDA without ϵ-loops, it performs a linear number of steps in the input length. At each step, at most N_A new branches are opened from each existing branch. Therefore, the total number of branches is $O({N_A}^n)$, where n is the number of input letters read. It follows that a DSAPDA can perform an exponential number of steps in the length of the input.

To achieve linear-time membership for DSAPDA languages, we must circumvent the potentially exponential number of stack branches that the automaton can open. To do so, we require the following immediate lemmas.

[4] Up to permutations on the order in which the branches were chosen for transitions.

Lemma 1. *During the computation, at any given time, there are at most M_A different state and stack-symbol configurations among the heads of the stack-branches of the automaton.*

Lemma 2. *If two branches have the same stack-head configuration, then they behave identically on the same input, as long as the stack height does not dip below the initial height of the head.*[5]

By these two lemmas, we do not need the full exponential power of SAPDA to decide membership for DSAPDA. The core concept of the implementation is to execute the minimal number of branches necessary (at most M_A). When a number of branch heads have the same configuration, they are combined to be one head. The computation then continues on the merged branch, as long as its stack is not empty. Once the merged branch empties, the computation continues on the original branches. Thus, at any given computation step, at most M_A branch heads are necessary, and we achieve linear time. Note that this implementation yields a DAG structure rather than a tree, see Figure 2.

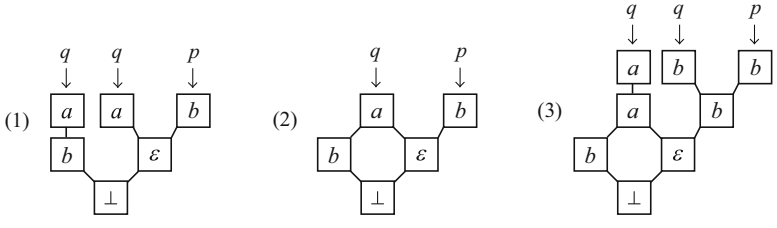

Fig. 2. Example of a stack structure in its initial state, after merging the two leftmost heads, and after applying a transition to the resulting structure

Theorem 1. *The membership problem for DSAPDA is decidable in linear time.*

The correctness of Theorem 1 stems from the observation that during each iteration, at most a constant number of stack symbols are added to the structure, and therefore it is always linearly bounded. For a full proof, see [2, Section 6.2, pp. 67–71].

Remark 2. There is a one-to-one correlation between the full configuration of a DSAPDA and its above compact representation, see [2, Remark 6.8, p. 71].

4 *LR*(0) Conjunctive Grammars

In this section, we extend the classical notion of an $LR(0)$ grammar (see [5, pp. 248–252]) to CG. We begin with some preliminary definitions.

Definition 7. *Let $G = (V, T, P, S)$ be a CG. The* trace grammar *of G is a* context-free *grammar $G_T = (V, T, P_T, S)$ where P_T is defined as follows:*

[5] This lemma stems from the fact that the automaton is *deterministic*.

1. $X \to \alpha \in P_T$ *for all* $X \to \alpha \in P$.
2. $X \to \alpha_i \in P_T$ *for all* $X \to (\alpha_i \& \cdots \& \alpha_n) \in P$ *and* $i = 1, \ldots, n$.

Rules of type 2 are called projections *of the original conjunctive rules they were obtained from. Applications of these rules are referred to as* conjunct selections.

Definition 8. *Let G be a conjunctive grammar, and let G_T be its trace grammar. A derivation in G_T is called a* trace *derivation of G. One-step trace derivations are denoted by $\Rightarrow_T$, and their closure by $\Rightarrow_T^*$.*

Definition 9. *Let G be a conjunctive grammar, and let G_T be its trace grammar. Let $\alpha \Rightarrow^* \mathcal{B}$ be a derivation in G, and let $\alpha \Rightarrow_T^* \beta$ be a trace derivation in G_T. We say that the trace derivation is a* projection *of the full derivation if it follows one of the possible conjunctive paths of the derivation, i.e., the conjunct selections in the trace match the conjunctive rules applied along the path of the full derivation, and the "regular" rules applied in the trace match the non-conjunctive rules applied in the path. If two trace derivations are both projections of the same full derivation, we say that they are* sibling *traces.*[6]

Consider the derivation of the word abc in the CG from Example 1. The traces $S \Rightarrow A \Rightarrow aA \Rightarrow aB \Rightarrow abBc \Rightarrow abc$ and $S \Rightarrow C \Rightarrow Cc \Rightarrow Dc \Rightarrow aDbc \Rightarrow abc$ are both projections of this derivation, and therefore siblings.

Definition 10. *A derivation $X \Rightarrow^* \mathcal{A}$ is a rightmost(leftmost) derivation if all its projections are rightmost(leftmost) in the classical sense. One-step rightmost(leftmost) derivations are denoted by $\Rightarrow_R (\Rightarrow_L)$, and their closure by $\Rightarrow_R^* (\Rightarrow_L^*)$.*

To facilitate in the construction of a parser, we define an augmented form for CG, which "marks" conjunctive rules using specified new variable symbols. The augmentation process is linear in the number of conjunctive rules in the grammar, and can be implemented as a simple pre-processing step in the construction of a parser.

Definition 11. *Given a conjunctive grammar $G = (V, \Sigma, P, S)$, we define an* augmented *grammar G' by adding the following variables and rules.*

- *We add a new start symbol S', and add a rule $S' \to S$.*
- *Let n be the maximal number of conjuncts in a rule of P. Let m be the number of conjunctive rules in P. For every rule $X \to (\alpha_1 \& \cdots \& \alpha_k) \in P$ that is the j-th conjunctive rule in P, we do the following.*
  - *Add new variables $S_1^j, \ldots, S_n^j$ and S^j,*
  - *replace the rule with $X \to S^j$, and add the rule $S^j \to (S_1^j \& \cdots \& S_n^j)$, and the rules $S_i^j \to \alpha_i$ for $i = 1, \ldots, k$ and $S_i^j \to \alpha_k$ for $i = k+1, \ldots, n$.*

Clearly, for any conjunctive grammar G and its augmentation G', $L(G') = L(G)$. Henceforth, we only consider augmented grammars.

[6] For an inductive definition of projections see [2, Definition 4.3, pp. 32–33].

Following is a very simple example of a conjunctive grammar in augmented form. The example will be used for illustrative purposes, to elucidate the parser construction.

Example 3. The following CG, $G = (V, \Sigma, P, S)$, is an augmented grammar which derives the regular language $\{ab\} \cup \{a^{6n}\$ \mid n \geq 1\}$.

- $V = \{S', S, A, B, S^1, S_1^1, S_2^1\}$, and
- P consists of the following rules:
  $S' \to S$; $S \to ab \mid S^1$; $S^1 \to (S_1^1 \ \& \ S_2^1)$; $S_1^1 \to aaA$; $S_2^1 \to aaaB$;
  $A \to aaA \mid \$$; $B \to aaaB \mid \$$

We proceed to define the basic building blocks of LR grammars, e.g., items, viable prefixes, and valid prefixes. We define these by applying the classical definitions to the trace grammar.

Definition 12. *Let $G = (V, \Sigma, P, S)$ be a CG. The* set of $LR(0)$ items *of G is the set of classical $LR(0)$ items of the trace grammar G_T,*[7] *i.e.,*

- $X \to \alpha \cdot \beta$ *is an $LR(0)$ item if $X \to \alpha\beta \in P$, and*
- $S^j \to \cdot \ S_i^j$ *and $S^j \to S_i^j \ \cdot$ s.t. $S^j \to (S_1^j \& \cdots \& S_n^j) \in P$ are $LR(0)$ items.*

Definition 13. *We say that $\gamma \in (V \cup \Sigma)^*$ is a* viable prefix *of G if it is a viable prefix of G_T, i.e., if there is a rightmost trace derivation of G_T of the form $S \Rightarrow_{TR}^* \delta X w \Rightarrow_{TR} \delta\beta w$ s.t. γ is a prefix of $\delta\beta$.*

Definition 14. *We say an item $X \to \alpha \cdot \beta$ is* valid *for a viable prefix γ if there is a trace derivation $S \Rightarrow_{TR}^* \delta X w \Rightarrow_{TR} \delta\alpha\beta w$, $X \to \alpha\beta \in P$, and $\gamma = \delta\alpha$.*

Example 4. Consider the augmented grammar G from Example 3. The derivation $S \Rightarrow S^1 \Rightarrow S_1^1 \Rightarrow aaA \Rightarrow aaaaA$ is a trace derivation of G. Therefore, all prefixes of $aaaaA$ are viable prefixes of G. It follows that the items $A \to a \cdot aA$ and $A \to aa \cdot A$ are valid for the viable prefixes aaa and $aaaa$ respectively.

We proceed to define a DSAPDA that acts as an LR parser for conjunctive languages. To do so, we define a canonical set of item-sets and two functions, *action* and *goto*, which together make up the parsing table for a given grammar. As in the classical case, *goto* recognizes valid items for viable prefixes, and *action* decides which step the automaton takes, based on the set of valid items supplied by *goto*. The main difference from the classical case is that when an item of the form $X \to \cdot \ S^j$ is valid (i.e., a conjunctive rule can be applied), the DSAPDA makes a conjunctive transition and each branch processes one of the conjuncts $S_1^j, \ldots, S_n^j$. This is to avoid conflicts in the parser caused by items from sibling traces.

We begin with *goto*. A *goto* function receives a set of items I and a symbol $X \in V \cup \Sigma \cup \{\epsilon\}$.[8] The function is applied to two types of item sets: *regular* and

[7] Note that the only assumption we make on G is that it is in augmented form.
[8] Note that in the classical *goto* function, $X \in V \cup \Sigma$.

split, see below. When a *goto* function g is applied to regular sets, it behaves exactly as in the classical case, i.e., if I is the set of valid items for some viable prefix γ and $X \neq \epsilon$ then $g(I, X)$ is the set of valid items for viable prefix γX.

When a *goto* function is applied to a split set, it only has ϵ-transitions. Namely, it has n ϵ-transitions to n item-sets, each containing exactly one of the $S^j \to \cdot\, S_i^j$ items, thus separating items from sibling traces. These transitions correlate with the conjunctive transitions in the DSAPDA parser. To accommodate for these "multiple transitions", the *goto* function maps to *sets* of item-sets, as opposed to exactly one item-set, as in the classical case.

Definition 15. *Let I be a set of $LR(0)$ items. We define the* item-closure *of I, denoted $[I]$ as the smallest set of items such that $I \subseteq [I]$, and if $X \to \delta \cdot Y\alpha \in [I]$ and $Y \neq S^j, j = 1, \cdots, m$, then $Y \to \cdot\, \beta \in [I]$ for all $Y \to \beta \in P$.*

This definition is the same as the classical item-closure definition, except that items of the form $X \to \cdot S^j$ are not expanded. This is because their expansions need to be separated, and they are therefore expanded in a separate step.

Definition 16. *A set of items I is* split *if it contains an item of the form $X \to \cdot\, S^j$, yet it does not contain the items $S^j \to \cdot\, S_i^j$. In this case, we also say that S^j* is split in *I. If an item set is not split, it is called* regular.

Note that by Definitions 15 and 16, the item-closure of a regular item-set corresponds to the classical item-closure definition.

Definition 17. *A function g is a valid goto function if for each regular item-set I, split item-set J, and symbol $X \in V \cup \Sigma$, the following holds.*

- $g(I, X) = \{\ [\{Z \to \alpha X \cdot \beta \mid Z \to \alpha \cdot X\beta \in I\}]\ \}$, *and* $g(I, \epsilon) = \{I\}$.
- $g(J, X) = \emptyset$, *and* $g(J, \epsilon) = \{[J_1], \ldots, [J_n]\}$ *where* $J_1, \cdots, J_n$ *are minimal item-sets such that*
  - $J \subseteq J_k$ *for* $k = 1, \ldots, n$, *and*
  - *for each j such that S^j is split in J, and for each $i = 1, \ldots, n$, there exists $1 \leq k \leq n$ such that $S^j \to \cdot S_i^j \in J_k$, and for no $i' \neq i$, $S^j \to \cdot S_{i'}^j \in J_k$.*

Note that in transitions from split-sets, exactly one $S^j \to \cdot\, S_i^j$ item from each conjunctive rule in J appears in each resulting item-set. In particular, if J contains only one item of the form $X \to \cdot S^j$, then for $k = 1, \ldots, n$, $J_k = [J \cup \{S^j \to \cdot S_{i_k}^j\}\ ]$, for some $1 \leq i_k \leq n$. Furthermore, note that when applied to regular sets, a valid *goto* function behaves exactly like the classical one.

Example 5. Continuing our discussion of the grammar G from Example 3, consider the item-set $I_0 = \{S' \to \cdot S, S \to \cdot ab, S \to \cdot S^1\}$. Note that I_0 is split. Therefore, a valid *goto* function can be defined by $g(I_0, \epsilon) = \{I_1, I_2\}$, where $I_1 = I_0 \cup \{S^1 \to \cdot S_1^1, S_1^1 \to \cdot aaA\}$, and $I_2 = I_0 \cup \{S^1 \to \cdot S_2^1, S_2^1 \to \cdot aaaB\}$. The sets I_1 and I_2 are regular. Therefore, e.g., as in the classical case, $g(I_1, a) = \{S \to a \cdot b, S_1^1 \to a \cdot aA\}$.

Next, we define the set of canonical item-sets of a conjunctive grammar, with respect to a valid *goto* function.

Definition 18. *Let $G = (V, \Sigma, P, S)$ be a conjunctive grammar and g a valid goto function. We define the* canonical collection of item-sets *of G with respect to g to be the smallest set C_g such that*

- $[\{S' \to \cdot\ S\}] \in C_g$, *and*
- *if $I \in C_g$ and $X \in V \cup \Sigma \cup \{\epsilon\}$, then $g(I, X) \subseteq C_g$.*

We denote the item-set $[\{S' \to \cdot\ S\}] \in C_g$ by I_0.

We now proceed to define the notion of an $LR(0)$ grammar.

Definition 19. *We say that an item-set I is* conflict free *if the following holds.*

- *If $X \to \alpha \cdot \in I$, then there is no other $Y \to \beta \cdot \in I$, $Y \neq X$ or $\alpha \neq \beta$.*
- *If $X \to \alpha \cdot \in I$, then there is no item $Y \to \beta \cdot \sigma\gamma \in I$, $\sigma \in \Sigma$.*

The first is a reduce-reduce conflict, and the second is a shift-reduce conflict.

Definition 20. *A conjunctive grammar $G = (V, \Sigma, P, S)$ is $LR(0)$ if there is a valid goto function g for which C_g is conflict free.*

Remark 3. Finding a conflict free grouping is a pre-processing step, and, therefore, does not impact the run-time of the parsing algorithm. Moreover, in Section 5, we will see that for every $LR(0)$ grammar, there exists an equivalent $LR(0)$ grammar, where *any* choice of grouping is guaranteed not to cause conflicts.

Let g be a valid *goto* function for a CG G, and let C_g be the resulting canonical set of item-sets. Together, g and C_g define a finite-state automaton where g is the transition function and C_g is the set of states. The automaton is non-deterministic, as g may have multiple ϵ-transitions from the same state. Let $\hat{g}$ be the standard extension of a non-deterministic transition function to strings and sets of states, see [5, pp. 24–25]. Then, for a set of item-sets $Q \subseteq C_g$ and a string $\gamma \in (V \cup \Sigma)^*$, $\hat{g}(Q, \gamma)$ contains all the item-sets J reachable from some set $I \in Q$ by reading γ.

Figure 3 describes a partial construction of the canonical set of item-sets and a valid *goto* function g for the grammar G from Example 3. Note that the first set, I_0 is split. Therefore, $g(I_0, \epsilon) = \{I_1, I_2\}$ where I_1 and I_2 each contain one of the items $S^1 \to \cdot S_1^1, S \to \cdot S_2^1$. Furthermore, we can see that, e.g.,

$$\hat{g}(\{I_0\}, a) = \{I_7, I_9\} =$$

$$\{\ \{S \to a \cdot b, S_1^1 \to a \cdot aA\}\ ,\ \{S \to a \cdot b, S_2^1 \to a \cdot aaB\}\ \}\ ,$$

and

$$\hat{g}(\{I_0\}, ab) = I_8 = \{S \to ab\ \cdot\}\ .$$

In the classical construction of an $LR(0)$ item automaton, $goto(I_0, \gamma)$ contains all the valid items for the viable prefix γ. We show that this also holds for a valid *goto* function of a conjunctive grammar. This stems from the fact that for

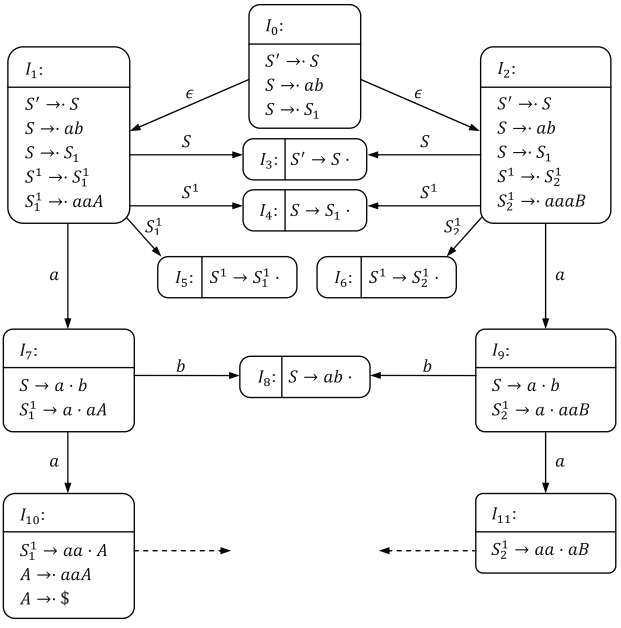

Fig. 3. Partial construction of the canonical set of item-sets and valid *goto* function

a conjunctive grammar G, the classical $LR(0)$ item-set automaton constructed for its trace, G_T, is the deterministic counterpart of our new item-set automaton construction applied to G. Therefore, the classical item-set automaton's capability to find valid items for viable prefixes is translated to the new construction as well, and we have the following lemma.

Lemma 3. *Let γ be a viable prefix of G and let g be a valid goto function. Then $\bigcup \hat{g}(\{I_0\}, \gamma)$ contains an item $X \to \alpha \cdot \beta$ if and only if $X \to \alpha \cdot \beta$ is valid for γ.*

We can now define a deterministic SAPDA that recognizes the language of an $LR(0)$ grammar G. For each branch, the automaton writes grammar symbols and item-sets from the canonical set of item-sets to its stack, thus keeping track of valid items for the viable prefix it has reduced so far. The transitions of the automaton are determined by the *action* function, and a valid *goto* function g. The initial symbol in the stack is the item-set I_0. The action function receives a current item-set from the top of the stack I, and the next symbol from the input $\sigma \in \Sigma$ (if such a symbol exists), and returns the following:

1. If I contains the item $S' \to S \cdot$, the stack is emptied (`accept`).
2. If I is regular and contains an item $X \to \alpha \cdot \sigma\beta$ then σ is shifted onto the stack, and $g(I, \sigma)$ is placed above it (`shift`). Note that $g(I, \sigma)$ returns a single item-set as a non-epsilon transition is applied.

3. If I is regular and contains an item $X \to \alpha \cdot$, then the symbols of α and the padding item-sets[9] are removed from the stack, revealing some item set J at the top of the stack. Now, X is written to the stack above J, and then $g(J, X)$ is written on top of that (`reduce`).
4. If I is split, then it is removed from the stack, n new branches are opened, and the n $g(I, \epsilon)$ item sets are put into them, one for each branch (`split`).

We now proceed to show that the automaton does, in fact, accept exactly the language of the grammar. As in the classical case, the automaton attempts to construct a rightmost derivation of the input. To do so, it stores the (tree) prefix of the derivation that it has managed to reduce so far in the stack. At each point, the top symbols of the branches in the stack are the item-sets obtained by applying the *goto* function to this prefix. By Lemma 3, these are exactly the valid items for the prefix, and therefore, they are the candidate derivation rules that can be added next to the rightmost derivation. The automaton continues to shift input symbols onto the stack, until the set of valid items contains an item of the form $X \to \alpha \cdot$. This signals that $X \to \alpha$ is the correct choice, and the symbols of α on the stack are replaced with X, thus simulating the reduction. Because the grammar is $LR(0)$, the item-sets are guaranteed to be conflict free, and therefore the automaton is well defined, i.e., it only has one valid transition defined for any given configuration. For a full proof that the rightmost derivation the automaton constructs is correct, see [2, pp. 80–83]. From our automaton construction, we have the following theorem.

Theorem 2. *If a language is generated by an $LR(0)$ conjunctive grammar, then it is accepted by a deterministic SAPDA.*

From Theorems 1 and 2, we obtain the following corollary.

Corollary 1. *Every $LR(0)$ conjunctive language can be parsed in linear time.*

This result extends the context-free $LR(0)$ algorithm, as $LR(0)$ conjunctive languages *properly* contain all finite intersections of classical $LR(0)$ languages. When applied to context-free $LR(0)$ grammars, the parsing algorithm is identical to the classical $LR(0)$ algorithm.

Remark 4. Both classical and conjunctive $LR(0)$ languages are not closed under complement (because the prefix property is not maintained) and under union. However, our linear parser can be modified to work for the boolean closure of conjunctive $LR(0)$ languages as follows. The complement of a language can be determined by running the parser and checking whether the final configuration is accepting or not. As the parser is deterministic, this method will correctly identify the words not in the language. The union of any finite number of languages can be recognized by simply making several parsing runs, one for each language. Thus, we have linear parsing for the boolean closure of $LR(0)$ conjunctive languages, and in particular, the boolean closure of classical $LR(0)$

[9] The padding item-sets are the item-sets pushed to the stack in type 2 transitions.

languages. Furthermore, in Proposition 1 at the end of the following section, we will see that conjunctive $LR(0)$ languages *strictly* contain the boolean closure of context-free $LR(0)$ languages.

5 Constructing an $LR(0)$ Grammar from a DSAPDA

In this section, we address the converse of Theorem 2, i.e., the construction of an $LR(0)$ conjunctive grammar from a deterministic SAPDA.

Theorem 3. *If a language is accepted by a deterministic SAPDA, then it is generated by an $LR(0)$ conjunctive grammar.*

Theorem 3 is proved similarly to the classical proof, see e.g., [5, pp. 256–260], by modifying the standard translation of an SAPDA into a CG. The constructed grammar has an important quality whereby for *every* possible valid *goto* function, the canonical set of item-sets constructed is conflict free. For the full construction and proof see [2, Section 6.4, pp. 84–96].

We conclude this section with the following proposition.

Proposition 1. *$LR(0)$ conjunctive languages contain a language that does not belong to the boolean closure of deterministic classical context-free languages.*

The proof of Proposition 1 can be derived directly from the fact that the language L_{inf} from Example 2 is not a finite intersection of context-free languages, see [2, Theorem 6.44, pp. 95–96].

In [11], Okhotin's generalized LR parsing algorithm promises linear-time parsing only for the boolean closure of context-free languages. As such, our result makes a stronger claim regarding the class of linearly-parsable languages.

In [10], Okhotin presents an $LL(k)$ parsing algorithm for conjunctive languages . In the paper, it is stated that it is an open question whether $LL(k)$ conjunctive grammars can generate a language which is not a finite intersection of context-free languages. Therefore, it is an open question whether languages such as L_{inf} can be generated by $LL(k)$ conjunctive grammars. The $LL(k)$ parsing algorithm utilizes a specialized tree-type structure as part of the parsing process. This tree-structure can be viewed as a special case of our SAPDA model, which aligns with the fact that the context-free versions of the $LL(k)$ algorithms is based on Pushdown Automata. Thus, it is reasonable to assume that the parsing algorithm could be modified to work with SAPDA, thus yielding a unified formal approach.

6 Concluding Remarks

We have introduced DSAPDA as a sub-family of SAPDA, and $LR(0)$ CG as a sub-family of CG, and shown that, as in the classical case, acceptance by a DSAPDA is equivalent to generation by an $LR(0)$ CG. This equivalence also forms the basis for a linear time parsing algorithm for an interesting language class comprised of the union closure of conjunctive $LR(0)$ languages.

It would prove interesting to define the notion of $LR(k)$ conjunctive grammars, and specifically $LR(1)$ conjunctive grammars. It would also be interesting to explore uses for DSAPDA based compilers. Two directions seem especially promising. The first is to look for examples where conjunctive grammars give a more succinct representation of a classical $LR(k)$ grammar, therefore leading to more efficient parsing. The second is to find examples of $LR(k)$ conjunctive languages which can be used to describe sophisticated constructs beyond the scope of context free languages. Such examples could prove useful for areas where context free languages have been known to be lacking, such as Natural Language Parsing.

References

1. Aizikowitz, T., Kaminski, M.: Conjunctive grammars and alternating pushdown automata. In: Hodges, W., de Queiroz, R. (eds.) Logic, Language, Information and Computation. LNCS (LNAI), vol. 5110, pp. 30–41. Springer, Heidelberg (2008)
2. Aizikowitz, T.: Synchronized Alternating Pushdown Automata. PhD thesis, Technion – Israel Institute of Technology (2010), `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2010/PHD/PHD-2010-14`
3. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM 28(1), 114–133 (1981)
4. Higginbotham, J.: English is not a context-free language. Linguistic Inquiry 15, 119–126 (1984)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
6. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8, 607–639 (1965)
7. Ladner, R.E., Lipton, R.J., Stockmeyer, L.J.: Alternating pushdown and stack automata. SIAM Journal on Computing 13(1), 135–155 (1984)
8. Langendoen, T.D., Postal, P.M.: English and the class of context-free languages. Computational Linguistics 10(3-4), 177–181 (1984)
9. Okhotin, A.: Conjunctive grammars. Journal of Automata, Languages and Combinatorics 6(4), 519–535 (2001)
10. Okhotin, A.: Top-down parsing of conjunctive languages. Grammars 5(1), 21–40 (2002)
11. Okhotin, A.: LR parsing for conjunctive grammars. Grammars 5(2), 21–40 (2002)
12. Okhotin, A.: A recognition and parsing algorithm for arbitrary conjunctive grammars. Theoretical Computer Science 302, 81–124 (2003)
13. Okhotin, A.: Fast parsing for boolean grammars: A generalization of valiant"s algorithm. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 340–351. Springer, Heidelberg (2010)
14. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)