AUTOMATA-BASED PARSING IN DYNAMIC PROGRAMMING FOR LINEAR INDEXED GRAMMARS

Miguel A. Alonso Pardo
Departamento de Computación, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
alonso@dc.fi.udc.es

Eric de la Clergerie
INRIA
Domaine de Voluceau, Rocquencourt, BP 105
78153 Le Chesnay Cedex, France
Eric.Clergerie@inria.fr

Manuel Vilares Ferro
Departamento de Computación, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
vilares@dc.fi.udc.es

Abstract

A general framework for the development of parsing algorithms in dynamic programming for Linear Indexed Grammars (LIG) is derived from the concept of Logic Push-down Automata (LPDA), an operational device for the construction of parsers for logic grammars. By exploiting several properties of the LIG formalism, we can guarantee both termination and completeness, which is not possible in the general case of logic grammars. In this paper we center our attention on the class of weakly predictive parsing strategies, which include bottom-up algorithms. The interpretation in dynamic programming of parsing algorithms belonging to this class can be performed in $\mathcal{O}(n^6)$ complexity, which is the lower bound achieved for LIG. In this context, a version for LIG of the LR parsing strategy is developed. The results are also applicable to other automata-based strategies, such as Left Corner.

Keywords: Linear Indexed Grammars, Logic Push-Down Automata, LR Parsing, Natural Language Processing.

This work was partially supported by Xunta de Galicia, (project XUGA10505B96) and Government of Spain (*Acción integrada* HF96-36). Part of this work was done while the first author was at INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.

INTRODUCTION

Tree Adjoining Grammars [14] (TAG) and Linear Indexed Grammars [5] (LIG) are the two most relevant members of the class of Mildly Context-sensitive Grammars (MCSG), which is placed between Context-free Grammars and Context Sensitive Grammars. The importance of TAG is due to its ability to express certain linguistic fenomena naturally. The relevance of LIG is due to its greater suitability for computational treatment. In fact, parsing algorithms for TAG and related formalisms often consider the compilation of the source grammar to an equivalent LIG [16, 10].

In recent years there has also been a great activity in the study of parsing techniques for logic grammar formalisms. Among the several devices developed with this purpose, we can notably cite Logic Push-down Automata [4] (LPDA) which constitute the basis of the general dynamic programming framework for the efficient implementation of resolution strategies for Horn-clause programs.

A specialized version of LPDA can be used as a general framework for describing parsing strategies for MCSG in dynamic programming, guaranteeing termination and completeness, which are achieved by using several properties derived from the lesser descriptive power of MCSG with respect to logic formalisms. In this paper, after describing the LIG formalism, we show how to translate LIG to DCG and how to interpret in dynamic programming the

LPDA that analyze the resulting grammars. A general framework for parsing algorithms belonging to the class of weakly predictive parsing strategies is described. In order to show the applicability of this framework, the LR parsing algorithm for Context-free Grammars is extended to LIG. The paper ends with a discussion of the work.

Linear Indexed Grammars

Indexed Grammars [1] are an extension of Context-free Grammars in which stacks of symbols are associated with non-terminals. Linear Indexed Grammars [5] are a restricted form of Indexed Grammars in which the dependences between stacks of non-terminals in each rule have been limited: at most one stack on the right-hand side is related with the stack of the left-hand side. The other stacks of the rule must have a bounded stack size.

Definition 1 *A LIG G is denoted by (V_N, V_T, V_I, P, S) where V_N,V_T,V_I and P denote finite sets of non-terminals, terminals, stack symbols and productions, and S is the start symbol of the grammar.* $\Box$

Following Gazdar [5], we will consider rules with the form:

$$\begin{array}{l} A[\circ\circ] \rightarrow \phi B[\circ\circ]\phi' \\ A[\circ\circ] \rightarrow \phi B[\gamma\circ\circ]\phi' \\ A[\gamma\circ\circ] \rightarrow \phi B[\circ\circ]\phi' \\ A[\,] \rightarrow w \end{array}$$

where $A, B \in V_N$, $w \in V_T$, $\gamma \in V_I$ and $\phi, \phi' \in (V_N[\,])^*$. Stacks are written with the top in the left-most position. In each rule, the element composed by B and its associate stack is called the *primary constituent* of the rule and elements in ϕ and ϕ' are called the *secondary constituents* of the rule.

LIG PARSING IN DYNAMIC PROGRAMMNG

From a grammatical point of view, LPDA are naturally adapted to parsing of Definite Clause Grammars [8] (DCG), a logical grammar formalism. All kind of parsing strategies for DCG can be described using LPDA, which are in fact a generalization of Push-Down Automata (PDA), the operational mechanism for CFG. Instead of single symbols, the elements of the stack in LPDA are logic atoms in some sets of predicate, function and variable symbols. In order to avoid confusion between the stack of LPDA and the stacks of LIG, we will write LPDA-stack for the former and LIG-stacks for the latter.

A very interesting property of LPDA is that they can be interpreted in dynamic programming. Because LPDA computations never use information below the top of the LPDA-stack until a pop action, computations can be factorized with respect to a bounded number of elements on the top of the LPDA-stack. Each of these pieces of information is called an *item* [4].

Considering that LIG-stacks can be represented as lists in DCG, as in the case of the programming language Prolog, we can define the following translation of LIG symbols to DCG, in which non-terminal symbols of LIG are unary predicates of DCG, LIG-stacks in the form of lists are the arguments of predicates, and variables are used to represent dependent parts of the LIG-stacks:

$$\begin{array}{lcl} A[\,] & \rightsquigarrow & A([\,]) \\ A[\circ\circ] & \rightsquigarrow & A(X) \\ A[\gamma\circ\circ] & \rightsquigarrow & A([\gamma \mid X]) \end{array}$$

This translation is extended to rules as follows, where ϕ_T represent the result of translating the elements in ϕ:

$$\begin{array}{llcl} & A[\circ\circ] & \rightarrow & \phi\, B[\circ\circ]\, \phi' \\ \rightsquigarrow & A(X) & \rightarrow & \phi_T\, B(X)\, \phi'_T \\ \\ & A[\gamma\circ\circ] & \rightarrow & \phi\, B[\circ\circ]\, \phi' \\ \rightsquigarrow & A([\gamma \mid X]) & \rightarrow & \phi_T\, B(X)\, \phi'_T \\ \\ & A[\circ\circ] & \rightarrow & \phi\, B[\gamma\circ\circ]\, \phi' \\ \rightsquigarrow & A\ (X) & \rightarrow & \phi_T\, B([\gamma \mid X])\, \phi'_T \\ \\ & A[\,] & \rightarrow & w \\ \rightsquigarrow & A([\,]) & \rightarrow & w \end{array}$$

Using this translation from LIG to DCG, LPDA can be used as an operational mechanism for LIG parsing. The LPDA-stack elements will be of the form $[V_N \cup V_T, \eta, i, j]$, where $\eta = [{V_I}^*] \cup [{V_I}^* \mid X]$, in order to represent the recognition of a terminal or non-terminal symbol and its associated LIG-stack between position i and j of the input string.

Every parsing algorithm developed for DCG using LPDA can be applied to LIG. Completeness is therefore guaranteed, as in the general case, but the DCG obtained from LIG may not have classical properties guaranteeing termination, like off-line parseability [8] or depth-boundness [6]. However, the termination can be ensured because each step in a derivation depends only on which of a finite set of "states" the derivation is in. This property, known as *context-freeness property* of LIG [15], implies that if the item

$$[A, [\gamma\delta\gamma \mid X], i, j]$$

can be derived from

$$[A, [\gamma \mid X], i, j]$$

then the set of items

$$\begin{array}{l}[A, [\gamma\delta\gamma\delta\gamma \mid X], i, j] \\ \vdots \\ [A, [\gamma\delta \ldots \gamma\delta\gamma \mid X], i, j]\end{array}$$

will also be derived. This set of items in fact represents the possibility of pushing the symbols $\gamma\delta'$ on the LIG-stack an unbounded number of times. Regular expressions can be used to finitely represent this kind of behavior, generating items such as

$$[A, [(\gamma\delta)^*\gamma \mid X], i, j]$$

and therefore allowing cycles to be represented in finite form. The use of regular expression in parsing strategies for MCSG was also suggested in [3], although in that case applied to TAG parsing. However, dealing with items in the previous form involves manipulating arbitrarily complicated regular expressions. Items can be simplified if we consider that:

1. Only the top element of a LIG-stack is considered in each derivation step.
2. Regular expressions, other than simple catenation of symbols, are only needed when there are cycles.

The first point suggest that we can think of items storing only the top element of a LIG-stack, but in order to access the rest of the LIG-stack[1] we also need a class of *pointer* that allows us to retrieve that information. Thus, items will be composed of two different parts: a *head*, which will store at least one element on the top of the LIG-stack associated to the top element of the LPDA, and a *rest*, which will point to the head of another item. Following the chain of rest pointers, you can retrieve the complete LIG-stack associated with an LPDA-stack element.

Bottom-up Parsing

In the case of CFG and DCG, weakly predictive parsing algorithms, that is algorithms in which the results of non-deterministic computations are constrained only by bottom-up propagation, can be interpreted in dynamic programming using items which only contain one element on the top of the LPDA-stack [4]. In the case of LIG, that element will be in the form of a *head*, composed of the non-terminal that has been recognized, the top element of its LIG-stack and two indexes of the input string, and a *rest* containing the head of the item storing the second element on the top of the LIG-stack:

$$[A, \gamma, i, j \mid A', \gamma', i', j']$$

In particular, a cycle that could be represented using items with regular expressions, such as $[A, [(\gamma\delta)^*\gamma\eta], i, j]$, will now be represented by the following set of items:

$$\begin{array}{ll} 0) & [A, \gamma, i, j \mid A', \eta_1, i', j'] \\ 1) & [A, \gamma, i, j \mid A_1, \delta_1, i, j] \\ 3) & [A_1, \delta_1, i, j \mid A_2, \delta_2, i, j] \\ & \vdots \\ n) & [A_n, \delta_n, i, j \mid A, \gamma, i, j] \end{array}$$

where the item 0 represents the state of the parsing process in which the symbol A has been recognized: the top of the LIG-stack associated to A is γ and the rest of the LIG-stack can be retrieved from the item with head A', η_1, i', j', with η_1 the top of η. The items from 1 to n represent the cycle by indicating in each step the symbol recognized, the top element of its LIG-stack and how to retrieve the rest of the LIG-stack.

The class of items studied in [15] for CYK-like parsing is very close to this class. This is not surprising, since CYK is a bottom-up algorithm based on dynamic programming, and is therefore constrained to using this kind of items.

LR PARSING FOR LIG

The class of LR parsing strategies constitutes one of the strongest and most efficient class of parsing strategies for Context-free Grammars. The class of grammars that can be deterministically analyzed using LR parsing with k lookahead symbols are called $LR(k)$ grammars [2]. They are useful for describing programming languages, but they are very limited when they are used for other purposes, for example parsing of natural languages. If we consider LR parsing tables in which each entry can contain several actions, we obtain non-deterministic LR parsing, often known as *generalized LR parsing*. A kind of generalized LR parsing was proposed by Tomita in [13]. It was implemented in dynamic programming using a *graph-structured stack* instead of a single stack in order to deal with multiple parses of a single sentence. Tomita's algorithm has problems with *cyclicity* and *hidden left recursive* constructions. Moreover, the complexity of Tomita's algorithm is $\mathcal{O}(n^{p+1})$, where p is the length of the longer right-hand side of a rule.

All kind of parsing algorithms for CFG can be described in the general framework for parsing laid down by Lang in [7], which is at the basis of dynamic programming interpretation of LPDA [4]. This framework was used in [17] to implement an efficient generalized LALR parsing algorithm for arbitrary Context-free Grammars which has $\mathcal{O}(n^3)$ as worst case complexity, regardless of the length of the rules. This parser was latter used as a backbone for developing the efficient parsing algorithm for DCG described in [18].

[1]In rules popping elements of the LIG-stack, we need to access the element below the top.

Construction of the LR automaton

In the case of LIG, as in the case of DCG, there are two possibilities for constructing of the LR automaton:

1. To consider only the context-free backbone of the Linear Indexed Grammar considered.
2. To include contextual information in the automaton.

The first option is simpler but it is less efficient because there is information in the grammar about the form of the LIG-stacks that is not considered during the construction of the automaton and therefore there will be more conflicts in the run-time phase of the algorithm. These conflicts could be avoided by including that kind of contextual information during the compilation of the LR automaton. In order to do that, we need to change the definition of the functions $first$ and $follow$ [2]:

Definition 2 *The function* **first**(Γ)*, with* $\Gamma \in V_T \cup V_N[V_I{}^*]$ *is defined as follows:*

1. *If* $\Gamma = a \in V_T$ $\mathbf{first}(a) = \{a\}$.
2. *If* $\Gamma \to \varepsilon$ *then* $\varepsilon \in \mathbf{first}(\Gamma)$.
3. *If* $\Gamma' \to \Gamma_1 \ldots \Gamma_i \ldots \Gamma_n \wedge \sigma = mgu(\Gamma, \Gamma')$ *then* $\mathbf{first}(\Gamma_1\sigma) \subseteq \mathbf{first}(\Gamma) \wedge \forall_{j=1}^{j=i-1} \Gamma_j$ *such that* $\varepsilon \in \mathbf{first}(\Gamma_j)$, $\mathbf{first}(\Gamma_{j+1}\sigma) \subseteq \mathbf{first}(\Gamma)$

where mgu stands for more general unifier. □

Definition 3 *The function* **follow**(Γ)*, with* $\Gamma \in V_N[V_I{}^*]$ *is computed using the following rules:*

1. *If* $\Gamma = S[\ ]$ *then* $\$ \in \mathbf{follow}(\Gamma)$*, where* $\$$ *is the end-marker of the input string and* S *is the axiom of the grammar.*
2. *If* $\Gamma'' \to \phi\Gamma'\phi' \wedge \sigma = mgu(\Gamma, \Gamma')$ *then* $\mathbf{first}(\phi'\sigma) \subseteq \mathbf{follow}(\Gamma)$.
3. *If* $\Gamma'' \to \phi\Gamma'\phi' \wedge \sigma = mgu(\Gamma, \Gamma') \wedge \varepsilon \in \mathbf{first}(\phi'\sigma)$ *then* $\mathbf{follow}(\Gamma''\sigma) \subseteq \mathbf{follow}(\Gamma)$. □

The closure of the states in the LR automaton is computed as in [2] except for the use of the new definition of the functions **first** and **follow**. As the closure operation is predictive in essence, problems can arise due to non-termination, because unification can produce an infinite number of new elements to be considered. This is a well known problem in logic grammars, a field in which several solutions have been proposed. Among these, the best adapted to LIG is the application of *restriction* as proposed by Shieber in [12]. In effect, as we only need to consider the top element of each LIG-stack in order to determine whether one rule can be used in a derivation or not, we can restrict the scope of the unification to the first element of the LIG-stack, considering the rest as a logic variable. Using this technique, only a finite set of elements will need to be considered during the compilation of the automaton. Once the automaton is constructed, the tables *goto* and *action* are computed as in [2].

Interpretation of the LR automaton in Dynamic Programming

As a first step, we must include a new element in the *head* and *rest* part of items: the state which the LR automaton is in when a grammatical symbol is recognized. The new form of items is

$$[A, \gamma, st, i, j \mid A', \gamma', st', i', j']$$

which means that non-terminal A with γ as top of its LIG-stack has been recognized in state st and it expands the part of the input string between positions i and j. The rest of the LIG-stack can be retrieved from items having $(A', \gamma', st', i', j')$ as *head*.

The initial item is $[S, \varepsilon, st_0, 0, 0 \mid -, -, -, -, -]$. For each item, we must check if some of the following operations can be performed:

1. *Shift.* If there exists an item $[A, \gamma, st, i, j \mid A', \gamma', st', i', j']$ and the corresponding action is a shift with terminal a_j to state st'', we will create the item $[a, \varepsilon, st'', j, j+1 \mid -, -, -, -, -]$.
2. *Reduction.* If there exists an item $[A, \gamma, st, i, j \mid A', \gamma', st', i', j']$ and the corresponding action is a reduction by rule r, we will create the item $[\nabla_{r,n_r}, \varepsilon, st, j, j \mid -, -, -, -, -]$, where n_r is the length of the right-hand side of rule r. Following [18], Symbols ∇ are used to indicated the part of the rule that has been reduced and $A_{r,s}$ will be used to indicate the symbol in position s in the right-hand side of rule r. Then the following steps are applied:
   (a) *Reduction-step.* For each item $[\nabla_{r,s}, \gamma, st, i, j \mid A', \gamma', st', i', j']$, if $A_{r,s}$ is not the primary constituent of rule r, we will seek an item $[A_{r,s}, \varepsilon, st, k, i \mid -, -, -, -, -]$ and we will generate the item $[\nabla_{r,s-1}, \varepsilon, st'', k, j \mid A', \gamma', st', i', j']$, such that $st \in goto(st'', A_{r,s})$. If $A_{r,s}$ is the primary constituent of the rule r, the item involved must be of the form $[\nabla_{r,s}, \varepsilon, st, i, j \mid -, -, -, -, -]$ and we will seek an item $[A_{r,s}, \gamma, st, k, j \mid B, \gamma_B, st_b, i_B, j_B]$. If the application of the rule involves a push on the LIG-stack, the item generated will be $[\nabla_{r,s-1}, \gamma', st'', k, j \mid A_{r,s}, \gamma, st, k, j]$, where γ' is the element pushed on the LIG-stack. If the application of the rule involves a pop in the LIG-stack, we will must follow the rest

pointer in order to retrieve the rest part $(C, \gamma_C, st_C, i_C, j_C)$ of the second element below the top of the LIG-stack. This last case gives the $\mathcal{O}(n^6)$ worst case complexity. We will then generate the item $[\nabla_{r,s-1}, \gamma_B, st'', k, j \mid C, \gamma_C, st_C, i_C, j_C]$.

(b) *Head-step.* If the item $[\nabla_{r,0}, st, \gamma, i, j \mid A', \gamma', st', i', j']$ exists then the right-hand side of rule r has been completely reduced and we will generate the item $[A_{r,0}, \gamma, st'', i, j \mid A', \gamma', st', i', j']$, where $st'' \in goto(st, A_{r,0})$.

If the item $[S, \varepsilon, st_f, 0, n \mid -, -, -, -, -]$ is generated, where S is the axiom of the grammar, st_f is the final state of the LR automaton and n is the length of the input string, then the input string have been recognized.

CONCLUSION

We have shown how Logic Push-down Automata, an operational mechanism developed for describing resolution strategies in the domain of logic programming, can be tuned for the analysis of Linear Indexed Grammar, resulting in a general framework for the construction of efficient LIG parsing algorithms in dynamic programming. In this context, the LR algorithm is extended to LIG, modifying the construction of the automaton in order to include contextual information provided by the grammar. The technique can be generalized for other parsing algorithms that use a compiled automaton during the run-time phase, for example Left Corner (LC) parsing [9]. The construction of states of the LC automaton must be performed as is described in [9] except for the functions **first** and **follow**, which must be substituted for the respective functions introduced in this paper.

References

[1] Aho, Alfred V., "Indexed Grammars — An Extension of Context-Free Grammars", *Journal of the Association for Computer Machinery*, 15(4):647–671, 1968.

[2] Aho, Alfred V. and Ullman, Jeffrey D., *The Theory of Parsing, Translation and Compiling*, Prentice Hall, 1972.

[3] Barthélemy, François and Villemonte de la Clergerie, Eric, "Subsumption–oriented Push–Down Automata", *Proc. of PLILP'92*, pp. 100–114, Springer-Verlag, 1992.

[4] De la Clergerie, Eric and Lang, Bernard, "LPDA: Another look at Tabulation in Logic Programming", in Van Hentenryck (ed.), *Proc. of the 11th International Conference on Logic Programming (ICLP'94)*, pp. 470–486, MIT Press, 1994.

[5] Gazdar, Gerald, "Applicability of Indexed Grammars to Natural Languages", in U. Reyle and C. Rohrer (eds.), *Natural Language Parsing and Linguistic Theories*, pp. 69–94, D. Reidel Publishing Company, 1987.

[6] Haas, Andrew, "A Parsing Algorithm for Unification Grammar", *Computational Linguistics*, 15(4), 219–232, 1989.

[7] Lang, Bernard, "Towards a Uniform Formal Framework for Parsing", in M. Tomita (ed.), *Current Issues in Parsing Technology*, pp. 153–171, Kluwer Academic Publishers, 1991.

[8] Pereira, Fernando C. N. and Warren, David H. D., "Parsing as Deduction", *Proc. of the 21st Annual Meeting of the Association for Computational Linguistics*, pp. 137–144, 1983.

[9] Schabes, Yves, "Polynomial Time and Space Shift-Reduce Parsing for Arbitrary Context-free Grammars", *Proc. of the 29th Annual Meeting of the Association for Computational Linguistics*, pp. 197–204, Berkeley, CA, USA, 1991.

[10] Schabes, Yves, "Stochastic Lexicalized Tree-Adjoining Grammars", Proc. of Fifteenth International Conference on Computational Linguistics (COLING'92), pp. 426–432, Nantes, France, 1992.

[11] Schabes, Yves and Joshi, Aravind K., "An Earley-type Parsing Algorithm for Tree Adjoining Grammars", Proc. of 26th Annual Meeting of the Association for Computational Linguistics, Buffalo, N.Y., 1988.

[12] Shieber, Stuart M., "Using Restriction to Extend Parsing Algorithms for Complex-feature-based Formalisms", *Proc. of the 23th Annual Meeting of the Association for Computational Linguistics*, pp. 145–152, 1985.

[13] Tomita, M., *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, Boston, Massachussets, USA, 1986.

[14] Vijay-Shanker, K., *A Study of Tree Adjoining Grammars*, Ph.D. Thesis, University of Pennsylvania, 1987.

[15] Vijay-Shanker, K. and Weir, David J., "Parsing Some Constrained Grammar Formalisms", *Computational Linguistics*, 19(4):591–636, 1994.

[16] Vijay-Shanker, K. and Weir, David J., "Polynomial Parsing of Extensions of Context-Free Grammars", in M. Tomita (ed.), *Current Issues in Parsing Technology*, pp. 191–206, Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[17] Vilares, Manuel, *Efficient Incremental Parsing for Context-free Languages*, PhD. thesis, Université de Nice, 1992.

[18] Vilares Ferro, Manuel . and Alonso Pardo,

Miguel A., "An LALR Extension for DCGs in Dynamic Programming", in P. Lucio, M. Martelli and M. Navarro, *Proc. of AGP'96 Joint Conference on Declarative Programming*, pp. 79–88, San Sebastián, Spain, 1996.