An LALR Extension for DCGs in Dynamic Programming

Manuel Vilares Ferro   Miguel A. Alonso Pardo

Abstract

We propose a parsing model for DCGs. Our work embodies in a common frame a dynamic programming construction developed for logical push-down automata, and techniques that restrict the computation to a useful part of the search space, inspired by LALR parsing. Unlike preceding approaches, our proposal avoids backtracking in all cases, providing computational sharing and operational completeness for DCGs without functional symbols.

Key Words: DCG, Dynamic Programming, LALR Parsing, Push-Down Automata.

1 Introduction

The popularity of DCGs is often related to natural language processing. In comparison with other formalisms, they seem to be particularly well-suited to controlling the perspicuity with which linguistic phenomena may be understood and expressed in actual language descriptions. However, an adequate description does not guarantee operational efficiency, and computational tractability is required if we intend to use descriptions for mechanical processing. Though much research has been devoted to this subject, most of the usable work in practice deals with the reduction of backtracking phenomena when parsing. To attain this goal, authors often follow two different approaches:

- Reduce the search space, using control techniques. Here, we distinguish between dynamic techniques when control is performed by rewriting programs [1, 2], and static ones when control is applied by an external

M. Vilares is with the Computer Science Department, University of Corunna, Campus de Elviña s/n, 15071 A Coruña, Spain. E-mail: vilares@dc.fi.udc.es.

M. A. Alonso is currently at INRIA, Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France. E-mail: Miguel.Alonso-Pardo@inria.fr.

Work partially supported by the Autonomous Government of Galicia under project XUGA20403B95.

driver, which determines the action to be performed [3]. Some authors [4] try to integrate both strategies.

- Incorporate dynamic programming techniques [5, 6], which assure that all parses are made in parallel, eliminating backtracking processes.

Our goal is to combine the advantages of the preceding approaches, eliminating the drawbacks. We focus on three aspects: Firstly, improving the quality of sharing, by reducing dependence on the syntactic context. Secondly, avoiding extra evaluation work. Finally, reducing the search space by indexing the parse, and implementing a garbage collector facility. We chose as our operational model the logical push-down automaton (LPDA), a formal engine introduced by Lang in [5] that generalizes the dynamic programming aspects of earlier evaluation strategies.

In Section 2 of this paper, we introduce dynamic programming in LPDA's. Section 3 describes our evaluation scheme, as well as our dynamic programming framework. In Section 4 we analyze bounds for both time and space. Section 5 compares our work with preceding proposals. In Section 6, we include a general consideration of the quality of the system. Finally, Section 7 is a conclusion regarding the work presented.

2 The general framework

In essence, an LPDA is a push-down automaton that stores logical atoms and substitutions on its stack and uses unification to apply transitions. We consider a simple variation from the original notion [5]. Formally, an LPDA is a 7-tuple $\mathcal{A} = (\mathcal{X}, \mathcal{F}, \Sigma, \Delta, \$, \$_f, \Theta)$, where: $\mathcal{X}$ is a denumerable and ordered set of variables, $\mathcal{F}$ is a finite set of functional symbols, Σ is a finite set of extensional predicate symbols, Δ is a finite set of predicate symbols used to represent the literals stored in the stack, $\$$ is the *initial predicate*, $\$_f$ is the *final predicate*; and Θ is a finite set of *transitions* of three kinds:

- *Horizontal:* $B \longmapsto C\{A\}$. Applicable to the stack $E.\rho\ \xi$, iff there exists the *most general unifier* (mgu), $\sigma = \text{mgu}(E, B)$ such that $F\sigma = A\sigma$, for a fact F in the extensional database. We obtain the new stack $C\sigma.\rho\sigma\ \xi$.
- *Pop:* $BD \longmapsto C\{A\}$. Applicable to stacks of the form $E.\rho E'.\rho'\ \xi$, iff there is $\sigma = \text{mgu}(< E, E'\rho >, < B, D >)$, such that $F\sigma = A\sigma$, for a fact F in the extensional database. The result will be the new configuration $C\sigma.\rho'\rho\sigma\ \xi$.
- *Push:* $B \longmapsto CB\{A\}$. We can apply it to configurations $E.\rho\ \xi$, iff there is $\sigma = \text{mgu}(E, B)$, such that $F\sigma = A\sigma$, for a fact F in the extensional database. We obtain the stack $C\sigma.\sigma\ B.\rho\ \xi$.

where B,C and D are literals in the algebra of terms $T_\Delta[\mathcal{F} \cup \mathcal{X}]$ and A is in $T_\Sigma[\mathcal{F} \cup \mathcal{X}]$, representing a control condition. Henceforth, we shall talk about *stacks* or *configurations* to refer to finite sequences of pairs *literal/substitution* denoted by $A.\sigma$, with the top on the left.

A *dynamic programming interpretation* of an LPDA $\mathcal{A}$ is the systematic exploration of a search space, whose elements $[\xi]$ are the classes of an equivalence relation $\mathcal{R}$ on the stacks ξ, that we call *items*. To manipulate this space, we define an operator *Op* adapting the transitions in $\mathcal{A}$ to their use with items. We use the term *dynamic frame* [7, 8] to refer to pairs $(\mathcal{R}, Op)$, and we denote by S^T the standard dynamic frame where *Op* is the identity and each stack is an item. Whichever the dynamic frame is, it must verify three properties in relation to S^T: Firstly, each computation in S^T has its corresponding counterpart in the dynamic frame (cf. compatibility). Secondly, all final configuration in S^T has its corresponding counterpart in the dynamic frame (cf. completeness). Finally, every final configurations found in our dynamic frame must correspond to final ones in S^T (cf. correctness).

The parsing algorithm proceeds by building items from the initial configuration by applying transitions to existing ones until no new application is possible. To assure fairness and completeness, an equitable selection order must be established in the search space. To ignore redundant items a subsumption-based relation must be put into place.

3 Improving sharing and efficiency

Due to the non-determinism of DCGs, it is convenient to merge the search space as much as possible. This saves on the space needed to represent items, and also on their processing.

3.1 The dynamic frame S^1

We exploit the possibilities of dynamic programming taking S^1 as the dynamic frame [7, 8]. We take items of the form $[A.u] := \{A.u\xi\}$, which implies that stacks are represented by their top. So, we reduce to a maximum extent the dependence on the syntactic context. To replace this lack of information, during pops, we define the operator *Op* as follows:

- *Horizontal case:* $Op(B \longmapsto C)([A]) = [C\sigma]$, where $\sigma = \text{mgu}(A, B)$.
- *Pop case:* $Op(BD \longmapsto C)([A]) = \{D\sigma \longmapsto C\sigma\}$, where $\sigma = \text{mgu}(A, B)$, and $D\sigma \longmapsto C\sigma$ is the *dynamic transition* generated by the pop transition. This transition is applicable not only to the configuration resulting from

the first one, but also to those to be generated and which share the same syntactic structure.

- *Push case:* $Op(B \longmapsto CB)([A]) = [C\sigma]$, where $\sigma = \mathrm{mgu}(A, B)$.

3.2 Reducing the search space

We index the parse by string position. So, we limit the search space at the time of recovery, as parsing progresses, by deleting information relating to earlier string positions. This relies on the concept of *itemset* [9], for which we associate a set of items to each token in the input string, and which represents the state of the parsing process at that point of the scan.

We extend itemsets to include dynamic transitions and decrease the number of such structures generated. To do this, it is sufficient to consider itemsets as synchronization points, generating them one by one. So, dynamic transitions in an itemset are only necessary once an empty reduction has been performed and ambiguity arises in the scope of the itemset [7].

We attach to each item a *back pointer* to the itemset associated to the input symbol at which we began to look for that configuration of the LPDA, as well as a pointer to the current itemset. Items are now triples $[A, \mathrm{itemset}, \mathrm{back\text{-}pointer}]$, where $A \in T_{\Delta}[\mathcal{F} \cup \mathcal{X}]$.

3.3 The control strategy

S^1 guarantees the best sharing quality for a given evaluation scheme, but the choice of this scheme can perceptibly alter the results [7]. A balance between on the other hand computational and sharing efficiency, and on the other one parser size, is the best basis for deciding. We focus on LALR(1)-like methods, which have a moderate splitting state phenomenon, improving both sharing and efficiency.

To build the driver, we recover the context-free backbone $\mathcal{G}^f$ of the DCG $\mathcal{G}$. We obtain terminals from the extensional database and non-terminals from heads in the intensional one. Terms with the same functor, but a different number of arguments correspond to different symbols in $\mathcal{G}^f$. We then build the LALR(1) automaton, which is probably non-deterministic, for $\mathcal{G}^f$, that is adapted to context-free parsing in S^1 [7]. To communicate the driver and the logical engine, we augment items with the state in which the driver is found, to obtain quadruples $[A, \mathrm{itemset}, \mathrm{back\text{-}pointer}, \mathrm{state}]$. We illustrate this work with Dyck-language and with one type of brackets. Throughout the rest of

this paper, the following DCG is our running example:

$$\begin{array}{lll} \gamma_1: & s(\mathrm{nil}) & \rightarrow \varepsilon \\ \gamma_2: & s(s(\mathrm{T}_1,\mathrm{T}_2)) & \rightarrow s(\mathrm{T}_1)\ s(\mathrm{T}_2) \\ \gamma_3: & s(s([,\mathrm{T},])) & \rightarrow [\ s(\mathrm{T})\ ] \end{array}$$

In this case, $\mathcal{G}^f$ is given by the context-free rules:

$$\gamma_0^f: \ \Phi \rightarrow S \dashv \qquad \gamma_1^f: \ S \rightarrow \varepsilon \qquad \gamma_2^f: \ S \rightarrow S\,S \qquad \gamma_3^f: \ S \rightarrow [\ S\ ]$$

To control pops in a reduction, given a clause γ_k defined by $A_{k,0} \rightarrow A_{k,1}, \ldots, A_{k,n_k}$ in a DCG $\gamma_{1..m}$, we consider: The vector $\vec{T}_k$ of the variables occurring in γ_k, and the predicate symbol $\nabla_{k,i}$. An instance of $\nabla_{k,i}(\vec{T}_k)$ indicates that all literals from the ith literal in the body of γ_k have been proved. So, our evaluation scheme is given by the transitions:

$$\begin{array}{lll} 1. & [A_{k,n_k}, it, bp, st] & \longmapsto [\nabla_{k,n_k}(\vec{T}_k), it, it, st]\,[A_{k,n_k}, it, bp, st] \\ & & \quad \{\mathrm{action}(st, \mathrm{token}_{it}) = \mathrm{reduce}(\gamma_k^f)\} \\ 2. & [\nabla_{k,i}(\vec{T}_k), it, r, st_1] & \\ & [A_{k,i}, r, bp, st_2] & \longmapsto [\nabla_{k,i-1}(\vec{T}_k), it, bp, st_2] \\ & & \quad \{\mathrm{action}(st_2, \mathrm{token}_{it}) = \mathrm{shift}(st_1)\},\ i \in [1, n_k] \\ 3. & [\nabla_{k,0}(\vec{T}_k), it, bp, st] & \longmapsto [A_{k,0}, it, bp, st] \end{array}$$

for the reduction mode, and

$$\begin{array}{lll} 4. & [A_{k,i}, it, bp, st_1] & \longmapsto [A_{k,i+1}, it+1, it, st_2]\,[A_{k,i}, it, bp, st_1] \\ & & \quad \{\mathrm{action}(st_1, \mathrm{token}_{it}) = \mathrm{shift}(st_2)\},\ i \in [0, n_k) \\ 5. & [\$, 0, 0, 0] & \longmapsto [A_{k,0}, 0, 0, st]\,[\$, 0, 0, 0] \\ & & \quad \{\mathrm{action}(0, \mathrm{token}_0) = \mathrm{shift}(st)\} \end{array}$$

for the scanning one. Briefly, we can interpret these transitions as follows:

1. *Selection of a clause:* Select the clause γ_k whose head is to be proved; then push $\nabla_{k,n_k}(\vec{T}_k)$ on the stack to indicate that none of the body literals have yet been proved.
2. *Reduction of one body literal:* The position literal $\nabla_{k,i}(\vec{T}_k)$ indicates that all body literals of γ_k following the ith literal have been proved. Now, all stacks having $\mathrm{A}_{k,i}$ just below the top can be reduced and in consequence the position literal can be incremented.
3. *Termination of the proof of the head of clause γ_k:* The position literal $\nabla_{k,0}(\vec{T}_k)$ indicates that all literals in the body of γ_k have been proved. Hence, we can replace it on the stack by the head $A_{k,0}$ of the rule, since it has now been proved.

4. *Pushing literals:* The literal $A_{k,i+1}$ is pushed onto the stack, assuming that it will be needed in reverse order for the proof.
5. *Initial push transition:* The initial predicate will be only used in push transitions, and exclusively as the first step of the LPDA computation.

Correctness and completeness are easily obtained from [7, 8], based on these results for LALR(1) context-free parsing and bottom-up evaluation without functional symbols[1], both using S^1 as dynamic frame.

4 Complexity bounds

Unrestricted DCGs have Turing machine power. So, it is not at all obvious to give a useful notion of computational complexity. Following Pereira and Warren in [6], we differentiate between *online* and *offline* parsing algorithms according to constraints due to unification that are considered as soon as rules are applied, or as a supplementary filtering phase after a classic context-free parsing. Given that the *offline* case seems to be the only linguistically relevant one, at the same time as the parsing problem is decidable, we estimate the complexity of doing online unification for offline parsable grammars. Assuming an input string of length n, our algorithm takes a time $\mathcal{O}(n^3)$ and a space $\mathcal{O}(n^2)$, in the worst case. The reasons are:

- The number of variables to access in an item and their ranges are both bounded. Only the value for the back pointer depends on i, and it is bounded by n. In consequence, the number of items associated to the string position i is $\mathcal{O}(i)$, and the algorithm needs a space $\mathcal{O}(\Sigma_{i=0}^{n} i) = \mathcal{O}(n^2)$.
- Push and horizontal transitions each execute a bounded number of steps per item in any itemset, while pop ones can execute $\mathcal{O}(i)$ steps because they may have to add $\mathcal{O}(l)$ items for the itemset in the position l pointed back to. So, it takes a time $\mathcal{O}(i^2)$ for the itemset in the position i, in the worst case, and time complexity for a successful parsing, including online unification and subsumption checking is $\mathcal{O}(\Sigma_{i=0}^{n} i^2) = \mathcal{O}(n^3)$.

For the class of *bounded item grammars*, the number of items is bounded whichever the itemset is, and linear time and space on the length of the input string are attained. This has a practical sense because this class of grammars includes the LALR(1) family and, in consequence, linear parsing can be performed while local determinism is present.

[1] the general case is not always decidable [6].

5 A comparison with previous works

We shall compare our work with some of the most representative approaches based on inference systems for logic programs. We take as reference the work of Nilsson in [3, 2], Bancilhon et al. in [1], Lang and de la Clergerie in [5, 8], and Rosenblueth and Peralta in [4].

Nilsson [3], and Rosenblueth and Peralta [4] propose SLR(1)-like evaluators, more efficient than grammar oriented algorithms, as [6], because they drastically limit backtracking. The difference between both approaches is due to the form in which they incorporate the contextual information present in DCGs. Nilsson ignores it to generate the driver, delaying its consideration until reduction occurs. To avoid this, Rosenblueth and Peralta concentrate the contextual information into the clauses of the extensional database, which forces the rewriting the original DCG in a non trivial manner, while the application domain is restricted to fixed-mode DCGs. Both, Nilsson and Rosenblueth et al., work in S^T and no indexing technique is considered. In consequence, the sharing quality is low.

Control can also be introduced by a goal-oriented strategy, whose efficiency depends on the amount of significant work required to evaluate the control predicates introduced. This is the case of the Magic Set methods [1, 2], which disregard the sharing problem.

Lang and de la Clergerie exploit the dynamic programming construction of LPDAs, but they always consider state-less automata, which reduce the efficiency of evaluation. The size of the search space depends only on the evaluation scheme while the efficiency depends also, to a great extent, on the DCG and the corpus of sentences to be analized, such as has been proved by the first author of this work in [7].

Our proposal guarantees completeness and correctness, in the case of absence of functional symbols, for unrestricted DCGs. Control is given by an LALR(1) driver, with a moderate state splitting phenomenon and large deterministic domain. The dynamic programming construction avoids backtracking, and the dynamic frame S^1 assures optimal sharing for the evaluation scheme. The use of itemsets as synchronization structures facilitates the reduction of the search space and the detection of cycles, a point that none of the preceding authors touch.

6 Experimental results

We have selected a scenario that cannot be qualified as advantageous. We search for DCGs with the following characteristics: The language includes sentences with a high density of ambiguities, to prove the adequacy of the algorithm for the sharing of computations. The grammar should also tackle the problem of recursive evaluation, and the data contain cycles to prove the adaptation to this feature. Finally, garbage collection should not be trivial.

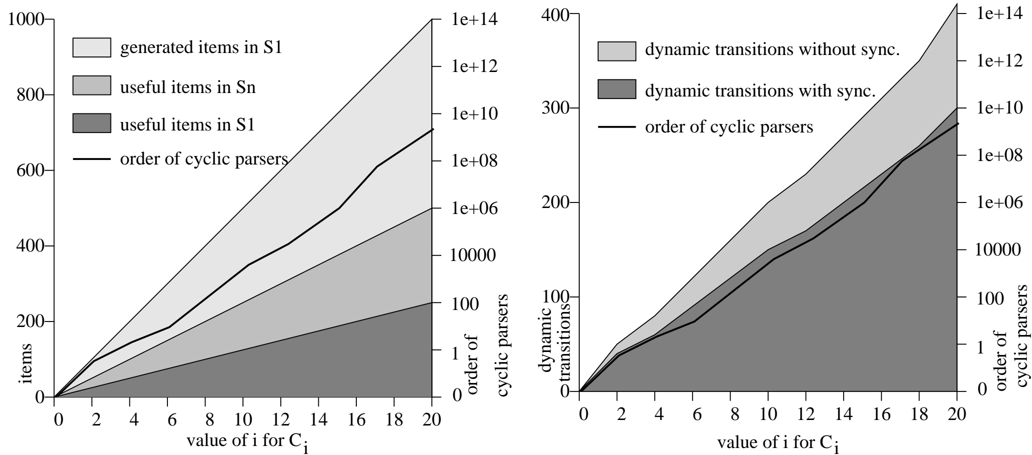

Figure 1: Number of items and dynamic transitions

In relation to these requirements, our running example seems to be a good candidate. Reductions imply, in the worst case, two terminals separated by another reduction. So, the use of indices does not allow large memory recovery. Taking as input strings sentences of the form $[\overset{i}{\ldots}[\,]\overset{i}{\ldots}]$, given that the grammar contains a rule $S \rightarrow S\ S$, the number of cyclic parses grows exponentially with i. This number is:

$$C_0 = C_1 = 1 \quad \text{and} \quad C_i = \begin{pmatrix} 2i \\ i \end{pmatrix} \frac{1}{i+1}, \text{ if } i > 1$$

On this basis, the scheme on the left in Fig. 1 gives the number of useful and useless items generated in S^1 and also compares the number of generated useful items in S^1 and S^T, while the right-hand scheme shows the number of dynamic transitions generated in S^1 considering synchronization on itemsets and also when that synchronization is not considered. Finally, the left-hand scheme in Fig. 2 shows the behavior of the garbage collector facility, while

the one on the right represents the number of unified pairs during the parse process. Additional costs due to the computation of the driver are irrelevant.

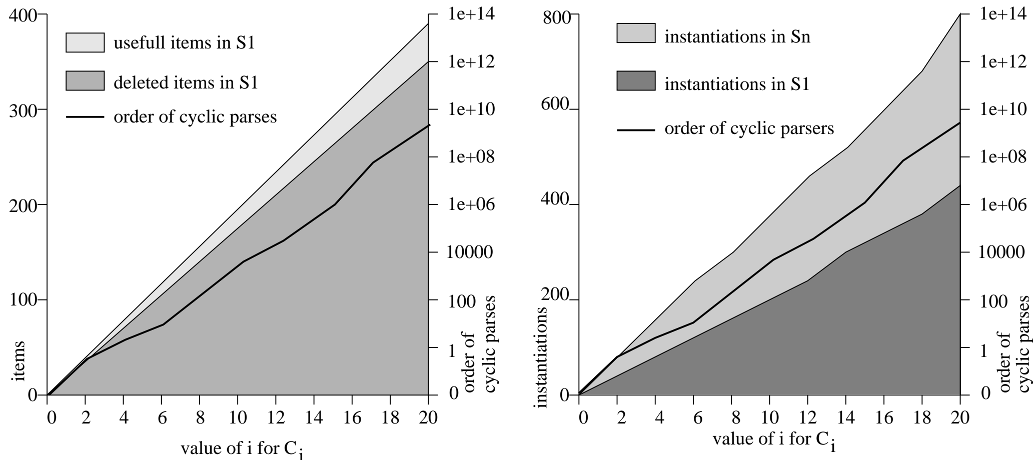

Figure 2: Unified pairs and garbage collector

We cannot really provide a comparison with other DCG parsers because of their problems in dealing with cyclic structures. We can however take results on S^T as a reference for non-dynamic SLR(1)-like methods [3, 4] since in this case the SLR(1) driver is closer to ours due to the small impact of the consideration of the lookahead facility. Naïve dynamic bottom-up methods [5, 8] can be assimilated to S^1 results without synchronization or garbage collector, also due to the small impact of state splitting in the example.

7 Conclusion

We have described a strategy for implementing DCG parsers. Our operational frame is an LPDA in dynamic programming. The architecture is a bottom-up evaluation scheme optimized with a predictive control provided by an LALR(1) driver. The system assures both an optimal treatment of sharing of computations, and completeness and correctness for DCGs without functional symbols.

Although preliminary results seem robust, there is still some work to be done to exploit the potential of our proposal to the fullest. In particular, future improvements will include an incremental facility in order to provide interactive parsing.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, "Magic-set and other strange ways to implement logic programs", in *Proc. of the 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[2] U. Nilsson, "Abstract interpretation: A kind of magic", in *Proc. of PLILP'91*, 1991.

[3] U. Nilsson, "AID: An alternative implementation of DCGs", *New Generation Computing*, vol. 4, pp. 383–399, 1986.

[4] D.A. Rosenblueth and J.C. Peralta, "LR inference: Inference systems for fixed-mode logic programs, based on LR parsing", in *International Logic Programming Symposium*, The MIT Press, Cambridge Massachussets 02142 USA, 1994, pp. 439–453.

[5] B. Lang, "Towards a uniform formal framework for parsing", in *Current Issues in Parsing Technology*, M. Tomita, Ed., pp. 153–171. Kluwer Academic Publishers, 1991.

[6] F.C.N. Pereira and D.H.D. Warren, "Parsing as deduction", in *Proc. of the* 21st *Annual Metting of the Association for Computational Linguistics*, 137-144, Ed., Cambridge, Massachusetts, U.S.A., 1983.

[7] M. Vilares Ferro, *Efficient Incremental Parsing for Context-Free Languages*, PhD thesis, University of Nice, France, 1992.

[8] E. Villemonte de la Clergerie, *Automates à Piles et Programmation Dynamique*, PhD thesis, University of Paris VII, France, 1993.

[9] J. Earley, "An efficient context-free parsing algorithm", *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.