An Operational Model for Parsing Fixed-Mode DCGs*

M. Vilares Ferro† M.A. Alonso Pardo† D. Cabrero Souto‡

Abstract

Logic programs share with context-free grammars a strong reliance on well-formedness conditions. Their proof procedures can be viewed as a generalization of context-free parsing. In particular, definite clause grammars can be interpreted as an extension of the classic context-free formalism where the notion of finite set of non-terminal symbols is generalized to a possibly infinite domain of directed graphs. In this case, standard polynomial parsing methods may no longer be applicable as they can lead to gross inefficiency or even non-termination for the algorithms.

We briefly present a proposal to avoid these drawbacks. We choose to work in the context of fixed-mode programs, focusing on two aspects: Avoiding limitations on the parsing process, and extending the unification to composed terms without overload for non-cyclic structures.

Key Words: DCG, Dynamic Programming, Logical Push-Down Automaton, Cyclic Term.

1 A parsing strategy for DCGs

Strategies for executing *definite clause grammars* (DCGs) are still often expressed directly as symbolic manipulations of terms and rules using backtracking, which does not constitute an adequate basis for efficient implementations. Some measures can be put into practice in order to make good these lacks: Firstly, to orientate the proof procedure towards a compiled architecture. Secondly, to improve the sharing quality of computations in a framework which is naturally not deterministic. Finally, to restrict the computation effort to the useful part of the search space.

Our operational formalism is an evolution of the notion of *logical push-down automaton* (LPDA) introduced by Lang in [1], a push-down automaton that stores logical atoms and substitutions on its stack, and uses unification to apply transitions. The *stack* is a finite sequence of *items* $[A, it, bp, st].\sigma$, where the top is on the left, A is a category in the DCG, σ a substitution, it is the current position in the input string, bp is the position in this input string at which we began to look for that configuration of $\mathcal{A}$, and st is a state for a driver controlling the evaluation. We choose as driver the LALR(1) automaton associated to the context-free skeleton of the logic grammar, by keeping only functors in the clauses to obtain terminals from the extensional database, and variables from heads in the intensional one.

To illustrate our work we consider as running example a simple DCG to deal with the sequentiation of nouns in English, as in the case of *"North Atlantic Treaty Organization"*. The clauses could be the following:

$$\text{s(X)} \ :\!-\text{np(X).} \qquad \text{np(np(Y,X))} \ :\!-\ \text{np(X) np(Y).} \qquad \text{np(X)} \ :\!-\ \text{noun(X:word).} \qquad \text{np(nil).}$$

In this case, the context-free skeleton is given by the context-free rules:

$$(0)\ \Phi \rightarrow \text{S} \dashv \qquad (1)\ \text{S} \rightarrow \text{NP} \qquad (2)\ \text{NP} \rightarrow \text{NP NP} \qquad (3)\ \text{NP} \rightarrow \text{noun} \qquad (4)\ \text{NP} \rightarrow \varepsilon$$

We exploit the possibilities of dynamic programming taking S^1 as dynamic frame [4, 5], by collapsing stacks on its top. In this way, we optimize sharing of computations in opposition to the

*Work partially supported by Government of Spain (HF96-36) and by Xunta de Galicia (XUGA10505B96).

†M. Vilares and M. A. Alonso are with Departamento de Computación, Universidad de La Coruña, Campus de Elviña s/n, 15071 La Coruña, Spain. E-mail: {vilares, alonso}@dc.fi.udc.es.

‡D. Cabrero is currently with Centro de Investigacións Lingüísticas e Literarias Ramón Piñeiro, Estrada Santiago-Noia, Km. 3, A Barcia, 15896 Santiago de Compostela, Spain. E-mail: dcabrero@cirp.es.

standard dynamic frame S^T, where stacks are represented by all their elements, or even S^2 using only the last two elements. In S^1 transitions are of three kinds, whose application over an item A is given by the following rules [5]:

- *Horizontal case:* $(B \longmapsto C)(A) = C\sigma$, where $\sigma = \text{mgu}(A, B)$.
- *Pop case:* $(BD \longmapsto C)(A) = \{D\sigma \longmapsto C\sigma\}$, where $\sigma = \text{mgu}(A, B)$, and $D\sigma \longmapsto C\sigma$ is the *dynamic transition* generated by the pop transition. This is applicable to the item resulting from the pop transition, and also probably to items to be generated.

  The number of dynamic transitions can be limited by grouping items in *itemsets* refereed to the analysis of a same word in the input string, and completing sequentially these itemsets. So, we can guarantee that a dynamic transition can be used to synchronize a computation to be done in this itemset if and only if the itemset is not locally deterministic and an empty reduction has been performed on it [4]. That establishes a simple criterion to save or not these transitions.
- *Push case:* $(B \longmapsto CB)(A) = C\sigma$, where $\sigma = \text{mgu}(A, B)$.

Given a DCG with clauses $\gamma_k : A_{k,0} :- A_{k,1}, \ldots, A_{k,n_k}$, we introduce the following notation: The vector $\vec{T_k}$ of the variables occurring in γ_k, and the predicate symbol $\bigtriangledown_{k,i}$, where an instance of $\bigtriangledown_{k,i}(\vec{T_k})$ indicates that all literals from the i^{th} literal in the body of γ_k have been proved. We can now describe the transitions:

1. $[A_{k,n_k}, it, bp, st] \longmapsto [\bigtriangledown_{k,n_k}(\vec{T_k}), it, it, st]\ [A_{k,n_k}, it, bp, st]$
   $\{\text{action}(st, \text{token}_{it}) = \text{reduce}(\gamma_k^f)\}$
2. $[\bigtriangledown_{k,i}(\vec{T_k}), it, r, st_1]$
   $[A_{k,i}, r, bp, st_1] \longmapsto [\bigtriangledown_{k,i-1}(\vec{T_k}), it, bp, st_2]$
   $\{\text{action}(st_2, \text{token}_{it}) = \text{shift}(st_1)\},\ i \in [1, n_k]$
3. $[\bigtriangledown_{k,0}(\vec{T_k}), it, bp, st_1] \longmapsto [A_{k,0}, it, bp, st_2]$
   $\{\text{goto}(st_1, A_{k,0}) = st_2\}$
4. $[A_{k,i}, it, bp, st_1] \longmapsto [A_{k,i+1}, it + 1, it, st_2]\ [A_{k,i}, it, bp, st_1]$
   $\{\text{action}(st_1, \text{token}_{it}) = \text{shift}(st_2)\},$
   $\text{token}_{it} = A_{k,i+1},\ i \in [0, n_k)$
5. $[A_{k,i}, it, bp, st_1] \longmapsto [A_{l,0}, it + 1, it, st_2]\ [A_{k,i}, it, bp, st_1]$
   $\{\text{action}(st_1, \text{token}_{it}) = \text{shift}(st_2)\},$
   $\text{token}_{it} = A_{l,0} \neq A_{k,i+1},\ i \in [0, n_k)$
6. $[\$, 0, 0, 0] \longmapsto [A_{k,0}, 0, 0, st]\ [\$, 0, 0, 0]$
   $\{\text{action}(0, \text{token}_0) = \text{shift}(st)\}$

where *action*(*state*, *token*) denotes the action of the driver for a given *state* and *token*, γ_k^f denotes the context-free rule in this driver corresponding to the clause γ_k, and expressions between brackets are conditions to be tested for the driver before applying transitions. Briefly, we can interpret these transitions as follows:

1. Select the clause γ_k whose head is to be proved, then push $\bigtriangledown_{k,n_k}(\vec{T_k})$ on the stack to indicate that none of the body literals have yet been proved.
2. The position literal $\bigtriangledown_{k,i}(\vec{T_k})$ indicates that all body literals of γ_k following the i^{th} literal have been proved. Now, all stacks having $\text{A}_{k,i}$ just below the top can be reduced and in consequence the position literal can be incremented.
3. The literal $\bigtriangledown_{k,0}(\vec{T_k})$ 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. The literal $A_{k,i+1}$ is pushed onto the stack, assuming that it will be needed for the proof.

5. The literal $A_{l,0}$ is pushed onto the stack in order to begin to prove the body of clause γ_l.

6. As a special case of the previous transition, the initial predicate will only be used in push transitions, and exclusively as the first step of the LPDA computation.

The parser build items from the initial one, applying transitions to existing ones until no new application is possible. An equitable selection order in the search space ensures fairness and completeness. Redundant items are ignored by a subsumption-based relation. Correctness and completeness, in the absence of functional symbols, are easily obtained from [4, 5], based on these results for LALR(1) context-free parsing and bottom-up evaluation, both using S^1 as dynamic frame. Our goal now is to extent these results to a larger class of grammars.

2 Extending unification to composed terms

To prevent the unification to loop, the concept of substitution is generalized to deal with cyclic terms, taking advantage of our fixed-mode orientation. As our bottom-up approach guarantees that at least one of the terms implied in a substitution is ground, the extension of the unification algorithm is limited to one case, when a predicate symbol is present. Here, after testing the compatibility of name and arity with the other term, the algorithm establishes if the associated non-terminals in the driver have been generated in the same state, covering the same portion of the text, which is equivalent to compare the corresponding back-pointers. If all these comparisons succeed, unification could be possible and we look for cyclicity, but only when these non-terminals show a cyclic behavior in the LALR(1) driver. In this case, the algorithm verifies, one by one, the possible occurrence of repeated terms by comparing the addresses of these with those of the arguments of the other predicate symbol. The optimal sharing of the interpretation guarantees that cyclicity arises if and only if any of these comparisons succeed. In this last case, the unification algorithm stops on the pair of arguments concerned, while continuing with the rest of the arguments.

Retaking our running example, we try to unify the terms in the box numbered 2 in Fig. 1, taking into account that we have previously unified terms in the box numbered 1. The terms to be unified are intermediate structures in the computation of the proof shared-forest associated to the successive reductions of rules 2 and 4 in the context-free skeleton. We compare the structures of the arguments associated to predicate symbol **"np"** using X, X', X'' and Y, Y', Y'' to denote the data structures corresponding to the variables and functors of the two terms in the example, and "$\rightarrow$" to denote a unification link from a represented symbol to its representative. Composed terms are denoted by a classic tree representation. The result of tree traversing is the cyclic structure shown in the box numbered 3.

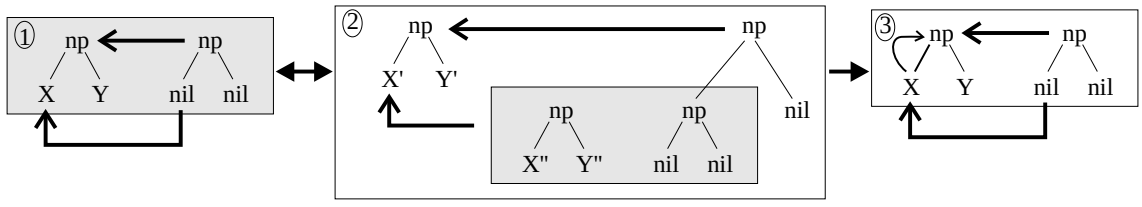

Figure 1: Cyclic tree traversing

Time complexity for the parser, including online unification and subsumption checking is in the worst case $\mathcal{O}(n^3)$ for input strings of length n. For bounded item grammars linear time and space 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.

3 Experimental results

For the tests we take our running example. Given that the grammar contains a rule NP → NP NP, the number of cyclic parses grows exponentially with the length, n, of the phrase. This number is:

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

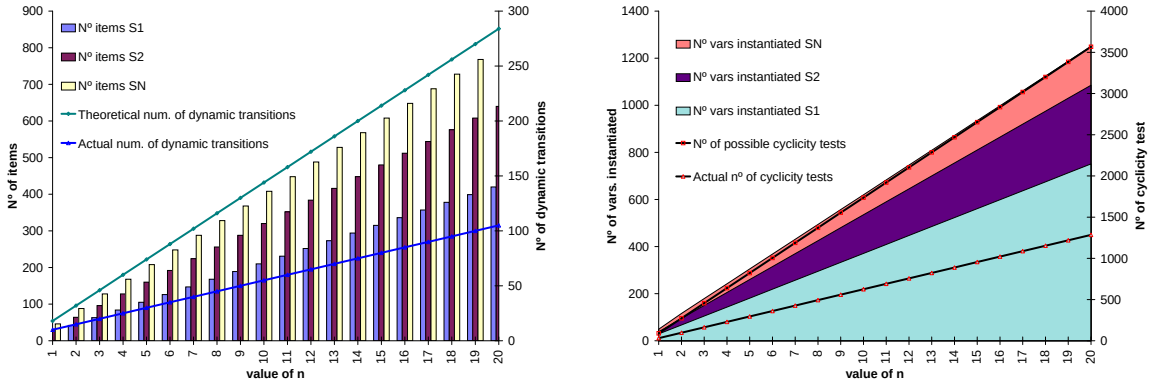

Figure 2: Some experimental results

We cannot really provide a comparison with other DCG parsers because of their problems in dealing with cyclic structures. We can however consider results on S^T as a reference for non-dynamic SLR(1)-like methods [2, 3], and naïve dynamic bottom-up methods [1, 5] can be assimilated to S^1 results without synchronization. This information is compiled in Fig. 3. The left-hand-side compares the generated items in S^1, S^2 and S^T, and the actual number of dynamic transitions generated in S^1 and the original number to be considered if no optimization is applied. The right-hand-side compares the variables instantiated in S^1, S^2 and S^T as well as the gain of computational efficiency due to the use of the LALR(1) driver for test cyclicity.

References

[1] B. Lang. Towards a uniform formal framework for parsing. In M. Tomita, editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publishers, 1991.

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

[3] 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*, pages 439–453, The MIT Press, Cambridge Massachussets 02142 USA, 1994.

[4] M. Vilares. *Efficient Incremental Parsing for Context-Free Languages*. PhD thesis, University of Nice. ISBN 2-7261-0768-0, France, 1992.

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