An Approach to Infinite Terms Traversal in DCGs*

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

Abstract

This paper describes a practical approach for detecting, traversing and representing cyclic terms in definite clause grammars. Our goal is to make a wider range of this formalism executable, without overload for non-cyclic structures. Unlike preceding approaches, our proposal avoids backtracking and changing of values in variables, increasing computational efficiency in a dynamic programming frame based on the logical push-down automaton model.

Key Words: Definite Clause Grammar, Dynamic Programming, Logical Push-Down Automaton, Cyclic Term.

1 Introduction

Grammar formalisms based on the encoding of grammatical information in unification-based strategies enjoy some currency both in linguistics and natural language processing. Such formalisms, as it is the case of *definite clause grammars* (DCGs), can be thought of, by analogy to context-free grammars, as generalizing the notion of non-terminal symbol from a finite domain of atomic elements to a possibly infinite domain of directed graph structures.

Although the use of infinite terms can be often avoided in practical applications, the potential offered by cyclic trees is fortly appreciated in language development tasks, where a large completion domain allows the modeling effort to be saved. Unfortunately, in moving to an infinite non-terminal domain, standard methods of parsing may no longer be applicable to the formalism. Typically, the problem manifests itself as gross inefficiency or even non-termination of the algorithms.

On the other hand, 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 two following approaches:

- To restrict in some way the parsing process. These limitations can be applied to the operational formalism by mandating a context-free backbone. Major category information in the original grammar is only used to filter spurious hypotheses by top-down filtering [1]. So, crucial information is often not used to eliminate useless computations.

  We can also couple grammar and parsing design, which is the case of some works based on constraint logic programming [11]. Since linguistic and technological problems are inherently mixed, this approach magnifies the difficulty of writing an adequate grammar-parser system.

  Finally, we can parametrize the parsing algorithm by grammar-dependent information that tells the algorithm which of the information in the feature structures is significant for guiding the parse [10]. Here, the choice for the exact parameter to be used is dependent

*Work partially supported by the Government of Spain under project HF96-36, and the Autonomous Government of Galicia under project XUGA10505B96.

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

‡D. Cabrero is currently with the Ramón Piñeiro Linguistic Research Center, Estrada Santiago-Noia, Km. 3, A Barcia, 15896 Santiago de Compostela, Spain. E-mail: dcabrero@cirp.es.

on both the grammar and the parsing algorithm, which produce results that are of no practical interest in a grammar development context.

- To extend the unification to provide the capacity to traverse cyclic trees. So, we can base unification on mechanisms other than resolution, avoiding occur checking. This is the case of Haridi and Sahlin in [3], who base unification on natural deduction [8]. Here, pointers are temporarily replaced in the structures, requiring undoing after execution, and unification of non-cyclic structures is penalized.

  It is also possible to modify the unification algorithm so that function and predicate symbols are treated in the same way as for variables, as is shown by Filgueiras in [2]. Essentially, the idea is the same as that considered by Haridi and Sahlin, and the drawbacks are extensible to this case.

  Finally, we can implement an algorithm for detecting and traversing cyclic trees by a simple generalization of cycle detecting algorithms for lists [4], as is the case of Nilsson and Tanaka in [6]. These strategies require some past nodes in structures to be saved and compared to new nodes. So, computational efficiency depends heavily on the depth of the structures.

Our goal is to combine the advantages of the preceding approaches, eliminating the drawbacks. We choose to work in the context of fixed-mode programs [9], that seem to be the only ones that are linguistically relevant, where each argument in a predicate acts either as input or as output of an operation. We shall focus on two aspects: Firstly, avoiding limitations on the parsing process. Secondly, extending the unification concept to composed terms without overload for non-cyclic structures.

In Section 2 of this paper, we present our parsing model for DCGs, a summary of the results described in [13] by the first and the third authors of this work. Section 3 describes our strategy for detecting and traversing cyclic terms. 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 A parsing model for DCGs

Strategies for executing DCGs are still often expressed directly as symbolic manipulations of terms and rules, which does not constitute an adequate basis for efficient implementations.

Sharing quality is another factor in obtaining efficiency in a framework which is not deterministic. This sharing saves on the space needed to represent the computations, and also on the later processing. Finally, it is also desirable to restrict computation effort to the useful part of the search space, which is not the case for analyzers based on backtracking.

2.1 The operational formalism

Our operational formalism is an evolution of the notion of *logical push-down automaton* (LPDA) introduced by Lang in [5], essentially a push-down automaton that stores logical atoms and substitutions on its stack, and uses unification to apply transitions.

For us, an LPDA is a 7-tupla $\mathcal{A} = (\mathcal{X}, \mathcal{F}, \Sigma, \Delta, \$, \$_f, \Theta)$, 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*. The *stack* of the automaton is a finite sequence of *items* $[A, it, bp, st].\sigma$, where the top is on the left, A is in the algebra of terms $T_\Delta[\mathcal{F} \cup \mathcal{X}]$, 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 the LPDA, and *st* is a state for the driver controlling the evaluation. Transitions are of three kinds:

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

where B, C and D are items and A is in $T_\Sigma[\mathcal{F} \cup \mathcal{X}]$, representing the control condition. The use of *it* and *bp* is equivalent to indexing the parse, which allows us to reduce the search space and to implement a garbage collector facility, by deleting information relating to earlier substrings, as parsing progresses. This relies on the concept of *itemset*, 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 illustrate this work with the DCG given by the following clauses:

$$\gamma_1: \ \text{s(X)} \ \rightarrow \ \text{f(X).} \qquad \gamma_2: \ \text{f(f(X))} \ \rightarrow \ \text{f(X).} \qquad \gamma_3: \ \text{f(a())} \ \rightarrow \ \text{a.}$$

throughout the rest of this paper, our running example

2.2 An LALR approach in dynamic programming

In order to maximize efficiency, we exploit the possibilities of dynamic programming taking S^1 as dynamic frame [12, 14] by collapsing stacks to obtain structures that we call *items*. Essentially, we represent a stack by its top. In this way, we optimize sharing of computations in opposition to the dynamic frames S^2, where the stack is collapsed in its last two items; and S^T, where stacks are represented by all their elements. To replace the lack of information about the rest of the stack during pop transitions, we redefine the behavior of transitions on items S^1, as follows:

- *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 not only to the item resulting from the pop transition, but also to those to be generated and which share the same syntactic context.
- *Push case:* $(B \longmapsto CB)(A) = C\sigma$, where $\sigma = \text{mgu}(A, B)$.

On the other hand, experience shows that the most efficient evaluation strategies seem to be those bottom-up approaches including a predictive phase in order to restrict the search space. So, our evaluation scheme is a bottom-up architecture optimized with a control provided by an LALR(1) driver, that we shall formalize now. Assuming a DCG of clauses $\gamma_k : A_{k,0} : -A_{k,1}, \ldots, A_{k,n_k}$, we introduce: The vector $\vec{T_k}$ of the variables occurring in γ_k, and the predicate symbol $\bigtriangledown_{k,i}$. 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 first recover the context-free skeleton of the logic program, by keeping only functors in the clauses to obtain terminals from the extensional database, and variables from heads in the intensional one. Terms with the same name, but a different number of arguments, correspond

to different symbols in the skeleton. In our running example, the context-free skeleton is given by the rules:

$$(0)\quad \Phi \rightarrow S\ \dashv \qquad (1)\quad S \rightarrow F \qquad (2)\quad F \rightarrow F \qquad (3)\quad F \rightarrow a$$

whose characteristic state machine is shown in Fig. 1. We consider now the following set of transitions:

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

for the reduction mode, and

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

for the scanning one, where $action(state, token)$ denotes the action of the LALR(1) automaton associated to the context-free skeleton, for a given *state* and *token*. 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 $\bigtriangledown_{k,n_k}(\vec{T_k})$ on the stack to indicate that none of the body literals have yet been proved.

2. *Reduction of one body literal:* The position literal $\bigtriangledown_{k,i}(\vec{T_k})$ indicates that all body literals of γ_k following the i^{th} literal have been proved. Now, all stacks having $A_{k,i}$ 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 $\bigtriangledown_{k,0}(\vec{T_k})$ 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. *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.

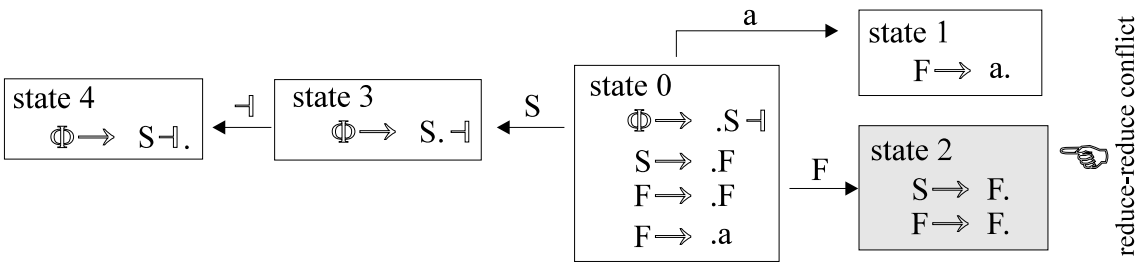

Figure 1: Characteristic state machine for the running example

The parsing algorithm proceeds by building items from the initial configuration, applying transitions to existing ones until no new application is possible. An equitable selection

order in the search space assures 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 [12, 14], based on these results for LALR(1) context-free parsing and bottom-up evaluation, both using S^1 as dynamic frame.

3 Traversing cyclic terms

We choose to separate cyclic tree traversing in two different phases: Cycle detection in the context-free backbone, and cycle traversing for predicate and function symbols by extending the unification algorithm to these terms. We justify this approach by the fact that the syntactic structure of the predicate symbols represents the context-free skeleton of the DCG. As a consequence, it is possible to efficiently guide the detection of cyclic predicate symbols on the basis of the dynamic programming interpretation for the LALR(1) driver. In effect, for cycles to arise in arguments, it is first necessary that the context-free backbone given by the predicate symbols determines the recognition of a same syntactic category without extra work for the scanning mode.

3.1 A necessary condition to loop

From the previous discussion, we can translate the first phase in our traverse strategy to detect cycles in context-free grammars in a dynamic frame S^1, using an LALR(1) parser.

This problem has previously been treated in [12] by the first author of this work, and the solution is very simple. Given that we have indexed the parse, it is sufficient to verify that in a same itemset the parsing process re-visits a state. In effect, this implies that an empty string has been parsed in a loop within the automaton. This can be shown on the context-free backbone of our running example. To do so, we consider the stack shared-forest in the left-hand-side of Fig. 2, where we include information about both the current state and the syntactic categories recognized. So, we can see that the reduce/reduce conflict at state 2 of Fig. 1 results in a cycle on the rule numbered 2.

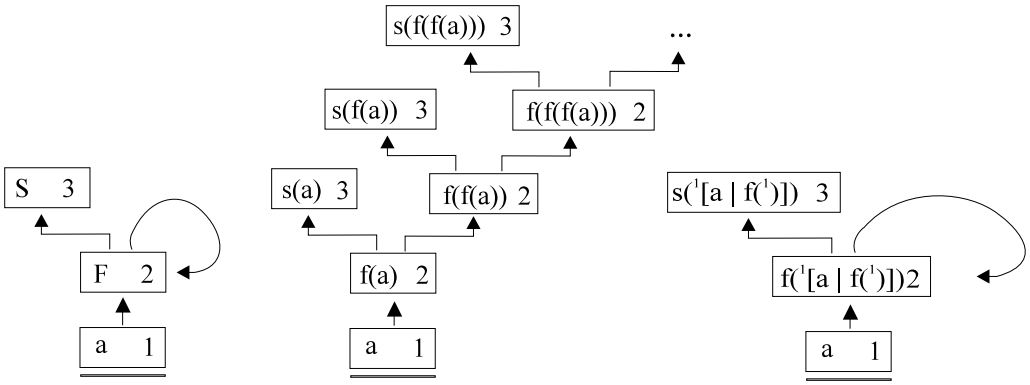

Figure 2: Stack shared-forests

To verify now that we can extend cyclicity to predicate symbols, it will be sufficient to test whether the terms implicated unify. In particular, we must generalize unification to detect cyclic terms with functional symbols.

It is important to remark that the condition explained can be performed in constant time, and therefore overload for non-cyclic structures is limited.

3.2 Extending unification

To prevent the unification to loop, the concept of substitution is generalized to include function and predicate symbol substitution. This means modifying the unification algorithm so that these symbols are treated in the same way as for variables.

Here, we take advantage of our fixed-mode orientation in a dynamic programming frame that guarantees an optimal sharing of computations. So, our bottom-up approach assures that at least one of the terms, cyclic or non-cyclic, implied in a substitution, is ground. This limits the complexity of the unification algorithm to the following three cases:

- There is an unbound variable, that will become bound to the ground term.
- There is a function symbol with unground arguments. In this case, the algorithm looks for the structural equivalence with the other term (ground), binding the unbound variables.
- There is a predicate symbol. Here, after testing the compatibility of name and arity with the other term, the algorithm establishes if the associated non-terminals in the LALR(1) driver have been generated in the same state, covering the same portion of the analyzed text[1]. At this point, we distinguish two cases:
  - If any of these comparisons does not succeed, unification is not possible. Both terms represent different nodes in the proof shared-forest.
  - If all these comparisons succeed, we can distinguish two further possibilities:
    * When arguments are the same for both predicate symbols, that implies the generation of a same node in the proof shared-forest in two different ways. So, we stop computations on this branch.
    * When arguments are not the same, we look for cycles in these, but only when the associated non-terminal to the predicate symbol in the LALR(1) driver shows a cyclic behavior.
      Here, 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 computations of the dynamic programming interpretation guarantees that cyclicity arises iff 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 the cyclicity previously detected in the context-free backbone of the running example, we check for the extension to the original DCG. We compare the structures of the arguments associated to predicate symbol **"f"** in the stack shared-forest in the center of Fig. 2, for the two upper occurrences. These are numbered 1 and 2 in Fig. 3, which illustrates the process resulting from the detection of the cycle numbered 3 in the figure. We use X, X', X'' 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, while composed terms are denoted by a classic tree representation. The right-hand-side of Fig. 2 shows the stack shared-forest after the cycle detection.

  Otherwise, unification does not succeed.

[1] this is equivalent to compare the corresponding back-pointers.

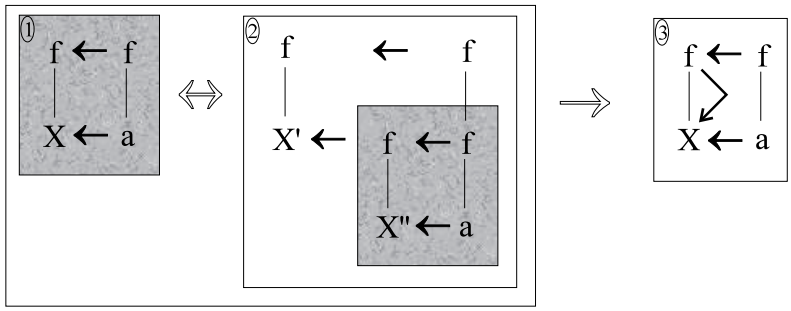

Figure 3: Tree traversal for a cyclic structure

4 Complexity bounds

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:

- For a given DCG, the depth of a cyclic term is bounded by the number $L * D$, where L is the maximum length of a cycle in its context-free skeleton, and D is the maximum depth of a function symbol.
- 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 regardless of the itemset, 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.

5 A comparison with previous works

In relation to systems forcing the primacy of major category [1], we only consider the context-free skeleton of a DCG as a guideline for parsing, without leaving out information about subcategorization. So, we apply constraints due to unification as soon as rules are applied, rather than considering a supplementary filtering phase after a classic context-free parsing.

On the other hand, the strategy described does not couple the design of descriptive and operational formalisms [11], nor even limits them [10]. In particular, we do not split up the infinite non-terminal domain into a finite set of equivalence classes that can be used for parsing. The only constraint is the consideration of fixed-mode DCGs, that we justify for their practical linguistic interest [9]. This allows us to conceive their consideration in a grammar development context.

In comparison with algorithms based on the temporary replacement of pointers in structures [3], our method does not need main memory references for pointer replacements. In addition, the absence of backtracking makes it unnecessary to undo work after execution, which facilitates the processing of shared structures.

Focusing now our attention on methods extending the concept of unification to composed terms [2], the overload for non-cyclic structures is often great. In our case, we minimize this factor of cost by a previous filtering phase to detect cyclicity in the context-free backbone. In the same way, the treatment of fixed-mode programs in a bottom-up evaluation scheme simplifies the unification protocol.

Finally, we can make reference to algorithms based on the memorization of nodes and comparison to new ones [6]. Here, the disadvantage is that these algorithms, to the best of our knowledge, cannot be optimized in order to avoid overload on non-cyclic structures.

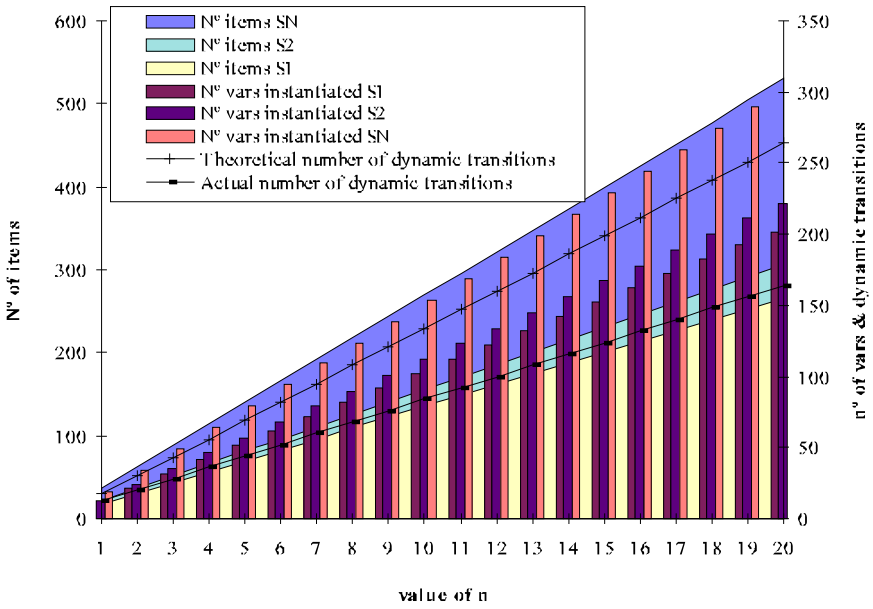

Figure 4: Number of items and instantiated variables

6 Experimental results

For the tests we looked for a DCG with the following characteristics that cannot be qualified as advantageous:

- 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.

We choose the Dyck-language with one type of brackets, given by the following clauses:

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

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

$$(0)\quad \Phi \rightarrow \mathrm{S}\ \dashv \qquad (1)\quad \mathrm{S} \rightarrow \varepsilon \qquad (2)\quad \mathrm{S} \rightarrow \mathrm{S}\,\mathrm{S} \qquad (3)\quad \mathrm{S} \rightarrow [\,\mathrm{S}\,]$$

As a consequence, taking as input strings sentences of the form $[.^n.\ [\ ]\ .^n.]$, given that the grammar contains a rule $S \rightarrow S\ S$, the number of cyclic parses grows exponentially with n. This number is:

$$C_0 = C_1 = 1 \quad \text{and} \quad C_n = \left(\begin{array}{c} 2n \\ n \end{array} \right) \frac{1}{n+1},\ \text{if } n > 1$$

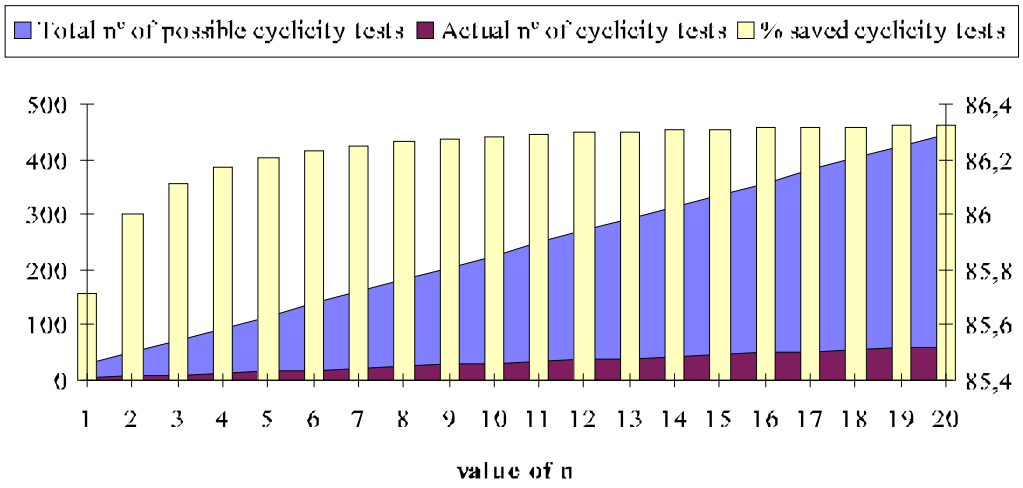

Figure 5: Economy of tests due to the use of the LALR(1) driver

We cannot really provide a comparison with other DCG parsers because of their problems in dealing with cyclic structures. Retaking the work of the first and third authors in [13], we can however consider results on S^T as a reference for non-dynamic SLR(1)-like methods [7, 9], and naïve dynamic bottom-up methods [5, 14] can be assimilated to S^1 results without synchronization. This information is compiled in Fig. 4: The number of items generated in S^1, comparing the generated items in S^1 and S^T, and the number of dynamic transitions generated in S^1 considering synchronization on itemsets as well as when that synchronization is not considered. There is also a comparison of the variables instantiated in S^1, S^2 and S^T.

In order to illustrate the gain of computational efficiency due to the use of the LALR(1) driver, Fig. 5 shows the number of tests to be performed for cyclic detection in both cases, using the LALR(1) driver and not.

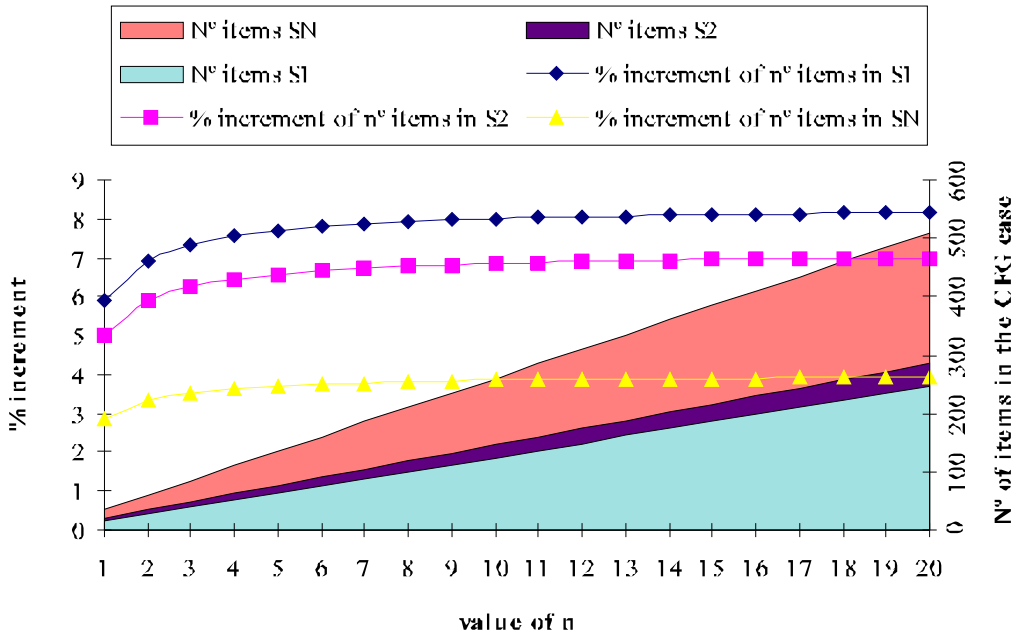

Figure 6: Relation between number of items in context-free and definite clause cases

In spite of giving a comparison with more simple formalisms, we have also tested the parser against itself in the context-free version [12], using the context-free backbone of the example grammar. The results are significant since the number of dynamic transitions is the same in

both cases and the increment in the number of items generated from the context-free parser to the definite clause parser is really small, as we can see in Fig. 6.

7 Conclusion

We have described a strategy for implementing efficient cyclic tree traversal in DCG parsers. Our operational frame is an LPDA in dynamic programming. The architecture is a parallel bottom-up evaluation scheme optimized with a predictive control provided by an LALR(1) driver, that avoids backtracking in all cases. The system assures both an optimal treatment of sharing of computations, and completeness and correctness for fixed-mode DCGs.

The motivation for the development of our proposal is inspired by the operational resemblance between classic context-free parsing and evaluation in first order Horn-logic, in particular in the case of DCGs. In this sense, we take advantage of the simplicity in the treatment of the context-free backbone of these logic programs to efficiently guide detection of cyclic structures. So, we significantly reduce the amount of work necessary for cyclic tree traversal, cutting down overload on non-cyclic structures.

References

[1] J. Bresnan and R. Kaplan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.

[2] M. Filgueiras. A PROLOG interpreter working with infinite terms. *Implementations of* PROLOG, 1984.

[3] S. Haridi and D. Sahlin. Efficient implementation of unification of cyclic structures. *Implementations of* PROLOG, 1985.

[4] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, Massachusetts, U.S.A., 1973.

[5] 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.

[6] M. Nilsson and H. Tanaka. Cyclic tree traversal. *LNCS*, 225:593–599, 1986.

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

[8] D. Prawitz. *Natural Deduction, Proof-Theoretical Study.* Almqvist & Wiksell, Stockholm, Sweden, 1965.

[9] 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.

[10] S.M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proc. of the 23th Annual Meeting of the ACL*, pages 145–152, 1985.

[11] F. Stolzenburg. Membership-constraints and some applications. Technical Report Fachberichte Informatik 5/94, Universität Koblenz-Landau, Koblenz, 1994.

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

[13] M. Vilares and M.A. Alonso. An LALR extension of DCG's in dynamic programming. In *Proc. of APPIA-GULP-PRODE'96 Joint Conference*, University of the Basque Country, San Sebastian, Spain, 1996.

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