Efficient Parsing of Fixed-Mode DCGs[*]

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

David Cabrero Souto
Centro de Investigacións "Ramón Piñeiro"
Estrada Santiago-Noia km. 3, A Barcia
15896 Santiago de Compostela, Spain
dcabrero@cirp.es

Abstract

In this paper we describe an efficient parsing model for monotonous fixed-mode DCGs, including traversing of cyclic terms. The algorithm can be viewed as an extension of the classic push-down automaton model in dynamic programming along with the incorporation of a mechanism for detecting and traversing cyclic trees. Details of implementation and experimental tests are described. Experimental results show the performance of our approach.

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

1 Introduction

One reason for the development of unification-based grammar formalisms has been that of having a programming environment for natural language processing. This is why logic programs resemble *context-free grammars*

[*]Supported by the Government of Spain under project HF96-36, and the Autonomous Government of Galicia under project XUGA10505B96.

(CFGs) and their proof procedures can be viewed as a generalization of context-free parsing [4]. We focus on *definite clause grammars* (DCGs), an extension of CFGs in which grammatical categories are replaced by Horn logic terms.

A relevant difference between CFGs and DCGs is given by the non-terminal domain, which is finite for the former and infinite for the latter. As a consequence, classic parsing techniques in CFGs may not guarantee termination when they are applied to DCGs. One approach guaranteeing termination that has been investigated is based on restricting to some extent the parsing process, for example mandating a non-cyclic context-free backbone [1], coupling grammar and parsing design [9] or parameterizing parsing algorithms with grammar dependent information [8]. Another approach has been to extend the unification in order to provide the capability of traversing cyclic trees, for example substituting resolution by another unification mechanism such as natural deduction [3], considering functions and predicates as elements with the same order as variables [2] or generalizing an available algorithm for traversing cyclic lists [5].

We try to combine the advantages provided for the above approaches eliminating, insofar as is possible, their drawbacks. We have chosen to work in the context of a restricted class of DCGs, known as *fixed-mode DCGs* [7], in which each argument in a predicate acts either as input or as output of an operation. This limitation does not seem to restrict the linguistic relevance of the grammars.

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 implementation.

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 [4], 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. 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.

In order to maximize efficiency, we exploit the possibilities of dynamic programming taking S^1 as dynamic frame [10, 12] 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 define 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 $\sigma = \text{mgu}(A, B)$.

where A, B, C and D are items.

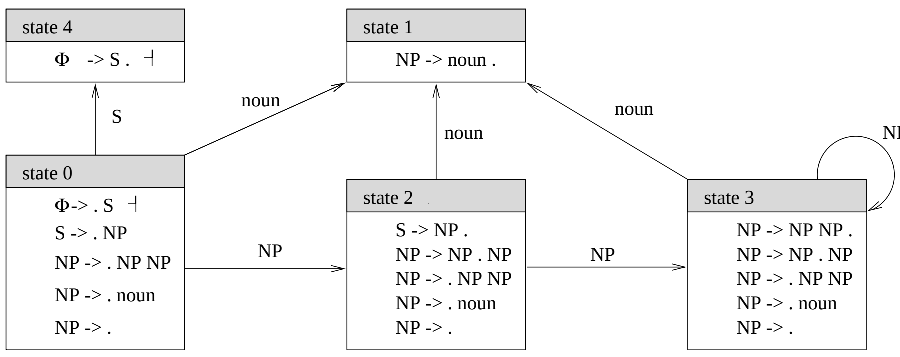

Figure 1: Characteristic finite state machine for the running example

2.2 An LALR approach in dynamic programming

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.

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, in which the arguments are used to build the abstract syntax tree, could be the following

(1) s(X) :−np(X).	(2) np(np(X, Y)) :− np(X) np(Y).
(3) np(X) :− noun(X:word).	(4) np(nil).

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

$$(0)\ \Phi \to S \dashv \qquad (1)\ S \to NP \qquad (2)\ NP \to NP\ NP$$
$$(3)\ NP \to noun \qquad (4)\ NP \to \varepsilon$$

whose characteristic finite state machine is shown in Fig. 1.

The parsing algorithm applies the following set of transitions:

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

for the reduction mode, and

$$
\begin{array}{lll}
4. & [A_{k,i}, it, bp, st_1] & \longmapsto \; [A_{k,i+1}, it+1, it, st_2]\; [A_{k,i}, it, bp, st_1] \\
 & & \qquad \{\text{action}(st_1, \text{token}_{it}) = \text{shift}(st_2)\}, \\
 & & \qquad \text{token}_{it} = A_{k,i+1},\; i \in [0, n_k) \\
5. & [A_{k,i}, it, bp, st_1] & \longmapsto \; [A_{l,0}, it+1, it, st_2]\; [A_{k,i}, it, bp, st_1] \\
 & & \qquad \{\text{action}(st_1, \text{token}_{it}) = \text{shift}(st_2)\}, \\
 & & \qquad \text{token}_{it} \neq A_{k,i+1},\; i \in [0, n_k) \\
6. & [\$, 0, 0, 0] & \longmapsto \; [A_{k,0}, 0, 0, st]\; [\$, 0, 0, 0] \\
 & & \qquad \{\text{action}(0, \text{token}_0) = \text{shift}(st)\}
\end{array}
$$

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 $\nabla_{k,n_k}(\vec{T_k})$ 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 i^{th} 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. *Pushing the first literal:* The literal $A_{l,0}$ is pushed onto the stack in order to begin to prove the body of clause γ_l.
6. *Initial push transition:* 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 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 [10, 12], based on these results for LALR(1) context-free parsing and bottom-up evaluation, both using S^1 as dynamic frame.

3 Parsing a sample sentence

To illustrate the algorithm, we are going to describe the parsing process for the simple sentence *North Atlantic* using our running grammar. From the initial predicate \$ on the top of the stack, and taking into account that the LALR automaton is in the initial state 0, the first action is the scanning of the word *North*, which involves pushing the item $[noun(\text{"}North\text{"}), 0, 1, st_1]$ that indicates the recognition of term $noun(\text{"}North\text{"})$ between positions 0 and 1 in the input string, with state 1 the current state in the LALR driver. This configuration is shown in Fig. 2.

At this point, we can apply transitions 1, 2 and 3 to reduce by clause γ_3. The configurations involved in this reduction are shown in Fig. 3.

We can now scan the word *Atlantic*, resulting in the recognizing of the term $noun(\text{"}Atlantic\text{"})$ between positions 1 and 2 in the input string, with the LALR driver in state 1. As in the case of the previous word, at this moment we can reduce by clause γ_3. This process is depicted in Fig. 4.

After having recognized two *np* predicates, we can reduce by clause γ_2 in order to obtain a new predicate *np* which will represent the nominal phrase *North Atlantic.* This reduction is shown in Fig. 5.

The recognition of the complete sentence ends with a reduction by clause γ_1, obtaining the predicate s, which has as argument the term

$$s(np(np(\text{"}North\text{"}), np(\text{"}Atlantic\text{"})))$$

representing the abstract parse tree for the sentence *North Atlantic.* The state of the LALR driver will now be 4, which is the final state, meaning

$$\begin{array}{|c|}\hline [\$,0,0,st_0] \\ \hline\hline \end{array} \vdash \begin{array}{|c|}\hline [noun(''North''),0,1,st_1] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

Figure 2: Configurations during the scanning of *North*.

$$\vdash \begin{array}{|c|}\hline [\nabla_{3,1}(X),1,1,st_1] \\ \hline [noun(''North''),0,1,st_1] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array} \quad \vdash \begin{array}{|c|}\hline [\nabla_{3,0}(''North''),0,1,st_0] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

Figure 3: Configuration during the reduction of clause γ_3.

$$\vdash \begin{array}{|c|}\hline [noun(''Atlantic''),1,2,st_1] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array} \quad \vdash \begin{array}{|c|}\hline [\nabla_{3,1}(X),2,2,st_1] \\ \hline [noun(''Atlantic''),1,2,st_1] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [\nabla_{3,0}(''Atlantic''),1,2,st_2] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array} \quad \vdash \begin{array}{|c|}\hline [np(np(''Atlantic'')),1,2,st_3] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

Figure 4: Configurations during the processing of the word *Atlantic*.

$$\vdash \begin{array}{|c|}\hline [\nabla_{2,2}(X,Y),2,2,st_3] \\ \hline [np(np(''Atlantic'')),1,2,st_3] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array} \quad \vdash \begin{array}{|c|}\hline [\nabla_{2,1}(X,np(''Atlantic'')),1,2,st_2] \\ \hline [np(np(''North'')),0,1,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [\nabla_{2,0}(np(''North''),np(''Atlantic'')),0,2,st_0] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [np(np(''North''),np(''Atlantic'')),0,2,st_2] \\ \hline [\$,0,0,st_0] \\ \hline\hline \end{array}$$

Figure 5: Recognition of the nominal phrase *North Atlantic*.

that the processing of this branch has finished. The resulting configurations are depicted in Fig. 6.

$$\vdash \begin{array}{|c|}\hline [\nabla_{1,1}(X), 2, 2, st_2] \\ \hline [np(np(''North''), np(''Atlantic'')), 0, 2, st_2] \\ \hline [\$, 0, 0, st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [\nabla_{1,0}(np(np(''North''), np(''Atlantic''))), 0, 2, st_0] \\ \hline [\$, 0, 0, st_0] \\ \hline\hline \end{array}$$

$$\vdash \begin{array}{|c|}\hline [s(np(np(''North''), np(''Atlantic''))), 0, 2, st_4] \\ \hline [\$, 0, 0, st_0] \\ \hline\hline \end{array}$$

Figure 6: Configurations for the recognizing of the sentence *North Atlantic.*

4 Extending unification to cyclic terms

Although structures that generate cyclic terms can be avoided in final systems, they usually arise during the development of grammars. For example, in the previous example we have shown the parsing process for only one branch, but the grammar really defines an infinite number of possible analyses for each input sentence. If we observe the LALR automaton, we can see that in states 0, 2 and 3 we can always reduce the clause γ_4, which has an empty right-hand side, in addition to other possible shift and reduce actions. In particular, in state 3 the predicate *np* can be generated an unbounded number of times without consuming any character of the input string.

Our parsing algorithm has no problems in dealing with non-determinism. It simply explores all possible alternatives in each point of the parsing process. This does not affect the level of sharing, which is achieved by the use of S^1 as dynamic frame, but it can pose problems with termination due to the presence of cyclic structures. Therefore, a special mechanism for representing cyclic terms must be used. At this point, it is important to remark that this mechanism should not decrease the efficiency in the treatment of non cyclic structures. In this context, we have separated cyclic tree traversal in two phases:

1. Cycle detection in the context-free backbone.

2. Cycle traversing for predicate and function symbols by extending the unification algorithm to these terms.

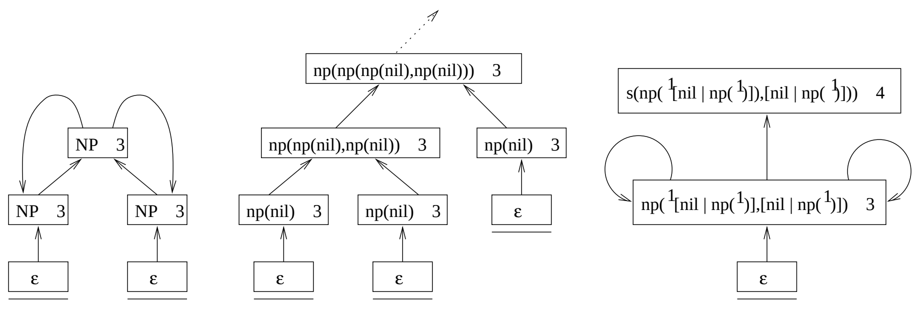
NP 3
NP 3
NP 3
ε
ε
np(np(np(nil),np(nil))) 3
np(np(nil),np(nil)) 3
np(nil) 3
np(nil) 3
np(nil) 3
ε
ε
ε
s(np(1[nil | np(1)]),[nil | np(1)])) 4
np(1[nil | np(1)],[nil | np(1)]) 3
ε

Figure 7: Cyclicities in the context-free skeleton and within terms.

The first phase [10] is performed verifying that in a given position in the input string, the parsing process re-visits a state. This implies that an empty string has been parsed in a loop within the automaton. Following with our example, we can see in the left-hand drawing of Fig. 7 a cycle in the context-free skeleton produced by successive reductions by rules 2 and 4 in state 3.

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

We take advantage both of working with monotonous fixed-mode grammars and of our bottom-up parsing algorithm, which guarantees that at least one of the terms implied in a substitution is ground. Then, the extension of the unification algorithm is limited to one case, namely 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 comparing 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 Fig. 7, once the context-free cyclicity has been detected, we check for possible cyclic term in the original DCG. The center drawing in that figure shows how the family of terms $np(np^*(nil), np(nil))$ is generated. In an analogous form, the family $np(np(nil), np^*(nil))$ can be generated, and in general the family $np(np^*(nil), np^*(nil))$ will be generated by successive applications of clauses γ_2 and γ_4. We now describe how we detect and represent these types of construction. In the first stages of the parsing process, two terms $np(nil)$ are generated, which are unified against $np(X, Y$, yielding $np(np(nil), np(nil))$. In the following stage, a unification will be tried between this last term and the variable X in $np(X, Y)$. At this point, we consider that:

- we are applying the same kind of unification as before, and
- there exists a cycle in the context-free backbone.

Therefore this process can be repeated an unbounded number of times for giving terms with the form $np(np^*(nil), np(nil))$. The same reasoning can be applied for the case of trying aq unification with the variable Y. The right-hand drawing in Fig.7 shows the compact representation we use in this case of cyclic terms. The functor np is considered in itself as a kind of special variable with two arguments. Each of these arguments can be either nil or a recursive application of np to itself. In the figure, superscripts are used to indicate where a functor is referenced by some of its arguments.

5 Experimental results

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

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

We cannot really provide a comparison with other DCG parsers because of their problems in dealing with cyclic structures. From our work in [11], we can however consider results on S^T as a reference for non-dynamic SLR(1)-like methods [6, 7], and naïve dynamic bottom-up methods [4, 12] can be

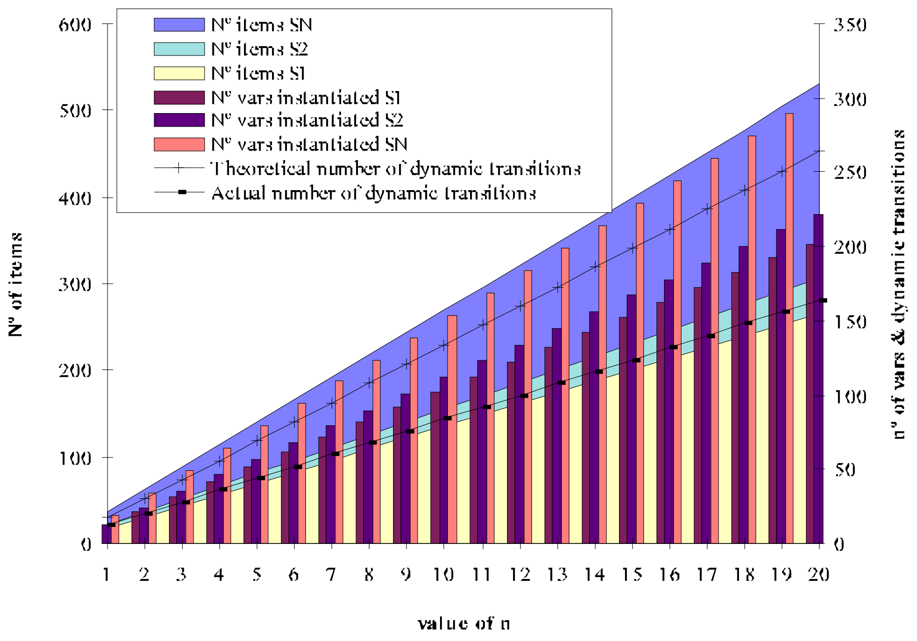

Figure 8: Number of items and instantiated variables

assimilated to S^1 results without synchronization. This information is compiled in Fig. 8: The number of items generated in S^1, comparing the items generated 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.

6 Conclusion

We have described an efficient strategy for analyzing DCG grammars which is based on a LPDA interpreted in dynamic programming, with a finite-state driver and a mechanism for dealing with cyclic terms. The evaluation scheme is parallel bottom-up without backtracking and it is optimized by predictive information provided by an LALR(1) driver. The system ensures a good level of sharing at the same time as it guarantees correctness and completeness in the case of monotonous fixed-mode DCGs. In this context, we exploit the context-free backbone of these logic programs to efficiently guide detection of cyclic constructions without overload for non-cyclic ones.

References

[1] J. Bresnan and R. Kaplan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan (ed.), *The Mental Representation of Grammatical Relations*, pp. 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] B. Lang. "Towards a uniform formal framework for parsing". In M. Tomita (ed.), *Current Issues in Parsing Technology*, pp. 153–171. Kluwer Academic Publishers, 1991.

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

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

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

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

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

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

[11] M. Vilares and M.A. Alonso. "An LALR extension of DCG's in dynamic programming". In C. Martín Vide (ed.), *Mathematical Linguistics, vol. II*, John Benjamins Publishing Company, Amsterdam, The Netherlands, 1997. To appear.

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