An Operational Model for Parsing Definite Clause Grammars with Infinite Terms

Manuel Vilares[1], Miguel A. Alonso[1], and David Cabrero[2]

[1] Department of Computer Sciences
Faculty of Informatics, University of A Coruña
Campus de Elviña s/n, 15071 A Coruña, Spain
e-mail: {vilares,alonso}@dc.fi.udc.es
[2] Ramón Piñeiro Research Center for Humanities
Estrada Santiago-Noia km 3, A Barcia,
15896 Santiago de Compostela, Spain
e-mail: dcabrero@cirp.es

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 present a proposal to avoid these drawbacks, focusing on two aspects: avoiding limitations on the parsing process, and extending the unification to composed terms without overload for non-cyclic structures.

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 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 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.

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 and using only major category information in the original grammar to filter spurious hypotheses by top-down filtering [1].
- coupling grammar and parsing design, which is the case of some works based on constraint logic programming [12]. Since linguistic and technological problems are inherently mixed, this approach magnifies the difficulty of writing an adequate grammar-parser system.
- parameterizing parsing algorithms with grammar dependent information, that tells the algorithm which of the information in the feature structures is significant for guiding the parse [11]. Here, the choice for the exact parameter to be used is dependent on both the grammar and the parsing algorithm, which produce results that are of no practical interest in a grammar development context.

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. This is the case of Haridi and Sahlin in [4], who base unification on natural deduction [9]. Here, pointers are temporarily replaced in the structures, requiring undoing after execution, and unification of non-cyclic structures is penalized.
- considering functions and predicates as elements with the same order as variables as is shown by Filgueiras in [3]. Essentially, the idea is the same as that considered by Haridi and Sahlin, and the drawbacks are extensible to this case.
- generalizing an available algorithm for traversing cyclic lists, as is the case of Nilsson and Tanaka in [7]. 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.

In Sect. 2 of this paper, we present our parsing model for DCGs, a summary of the results described in [14]. An example of parsing is shown in Sect. 3. Section 4 describes our strategy for detecting and traversing cyclic terms. Sect. 5 compares our work with preceding proposals. In Sect. 6 we characterize the application domain of the algorithm. Sect. 7, includes a general consideration of the quality of the system. Finally, Sect. 8 is a conclusion regarding the work presented.

2 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.

2.1 Logical Push-Down Automata as Operational Formalism

Our operational formalism is an evolution of the notion of *logical push-down automaton* (LPDA) introduced by Lang in [5], 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-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*. 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 $C\sigma.\rho\sigma\ \xi$.
- *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 $C\sigma.\sigma\ B.\rho\ \xi$.
- *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 $C\sigma.\rho'\rho\sigma\ \xi$.

where B, C and D are items, and A is in $T_\Sigma[\mathcal{F} \cup \mathcal{X}]$ representing the control condition.

In bottom-up evaluation strategies, we can exploit the possibilities of dynamic programming taking S^1 as dynamic frame [13, 2], by collapsing stacks on its top. In this way, we optimize sharing of computations in opposition to the standard dynamic frame S^T, where stacks are represented by all their elements, or even S^2 using only the last two elements. To replace the lack of information about the rest of the stack during pop transitions, we redefine the behavior of transitions on items I in a dynamic frame S^1, as follows:

- *horizontal case:* $(B \longmapsto C)(I) = C\sigma$, where $\sigma = \text{mgu}(I, B)$.
- *push case:* $(B \longmapsto CB)(I) = C\sigma$, where $\sigma = \text{mgu}(I, B)$.
- *pop case:* $(BD \longmapsto C)(I) = \{D\sigma \longmapsto C\sigma\}$, where $\sigma = \text{mgu}(I, B)$, 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* that refer to the analysis of a same word in the input string, and completing in sequence 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 [13]. That establishes a simple criterion to save or not these transitions.

where we have omitted the use of control conditions, $\{A\}$, in order to simplify the exposition.

2.2 LALR Parsing 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 $\boldsymbol{T}_k$ of the variables occurring in γ_k, and the predicate symbol $\bigtriangledown_{k,i}$. An instance of $\bigtriangledown_{k,i}(\boldsymbol{T}_k)$ indicates that all literals from the i^{th} literal in the body of γ_k have been proved.

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. We can now describe the transitions:

1. $[A_{k,n_k}, it, bp, st] \longmapsto [\bigtriangledown_{k,n_k}(\boldsymbol{T}_k), it, it, st]\ [A_{k,n_k}, it, bp, st]$
   $\{\text{action}(st, \text{token}_{it}) = \text{reduce}(\gamma_k^f)\}$
2. $[\bigtriangledown_{k,i}(\boldsymbol{T}_k), it, r, st_1]$
   $[A_{k,i}, r, bp, st_1] \longmapsto [\bigtriangledown_{k,i-1}(\boldsymbol{T}_k), it, bp, st_2]$
   $\{\text{action}(st_2, \text{token}_{it}) = \text{shift}(st_1)\},\ i \in [1, n_k]$
3. $[\bigtriangledown_{k,0}(\boldsymbol{T}_k), it, bp, st_1] \longmapsto [A_{k,0}, it, bp, st_2]$
   $\{\text{goto}(st_1, A_{k,0}) = st_2\}$
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:

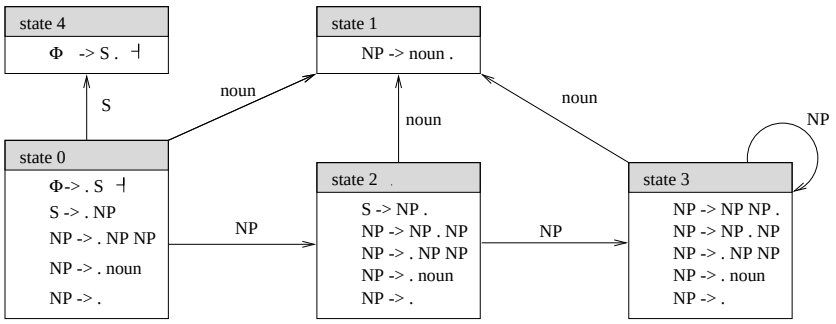

Fig. 1. Characteristic finite state machine for the running example

1. select the clause γ_k whose head is to be proved, then push $\bigtriangledown_{k,n_k}(\boldsymbol{T}_k)$ on the stack to indicate that none of the body literals have yet been proved.
2. the position literal $\bigtriangledown_{k,i}(\boldsymbol{T}_k)$ 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. the literal $\bigtriangledown_{k,0}(\boldsymbol{T}_k)$ 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 builds 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 [13, 2], 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.

3 Parsing a Sample Sentence

To illustrate our work we consider as running example a simple DCG to deal with the sequentialization 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

γ_1 : s(X) $\rightarrow$ np(X). γ_2 : np(np(X, Y)) $\rightarrow$ np(X) np(Y).
γ_3 : np(X) $\rightarrow$ noun(X). γ_4 : np(nil).

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

(0) Φ $\rightarrow$ S $\dashv$ (1) S $\rightarrow$ NP (2) NP $\rightarrow$ NP NP
(3) NP $\rightarrow$ noun (4) NP $\rightarrow$ ε

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

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.

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

Fig. 2. Configurations during the scanning of "*North*".

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.

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

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

Fig. 3. Configuration during the reduction of clause γ_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.

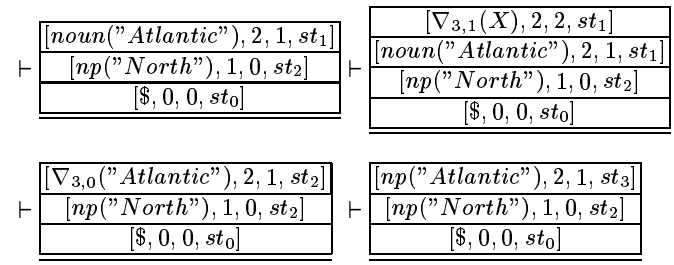

Fig. 4. Configurations during the processing of the word "*Atlantic*".

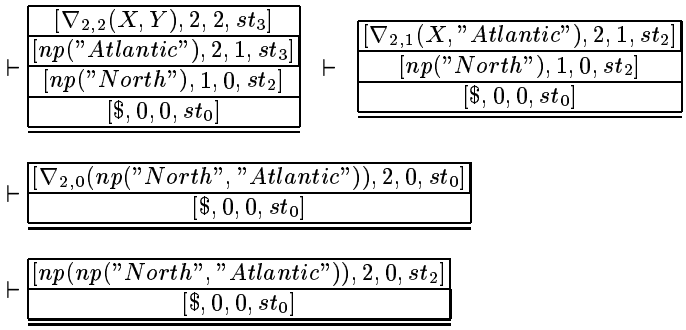

Fig. 5. Recognition of the nominal phrase "*North Atlantic*".

The recognition of the complete sentence ends with a reduction by clause γ_1, obtaining the term

$$s(np("North","Atlantic"))$$

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

4 Traversing 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.

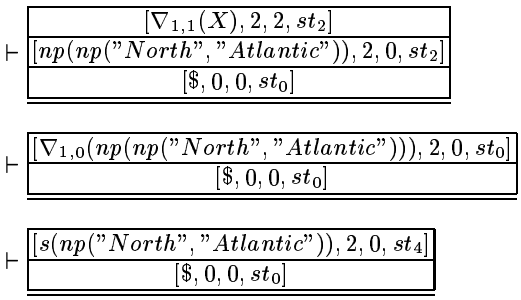

Fig. 6. Configurations for the recognizing of the sentence "*North Atlantic*".

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:

- cycle detection in the context-free backbone.
- 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. It should be pointed out that the reciprocal is not always true. This is, for example, the case of the DCG defined by the following clauses:

$$\gamma_1 : \ \mathrm{a(nil).} \qquad \gamma_2 : \ \mathrm{a(f(Y,X))} \ \rightarrow \ \mathrm{a(X).}$$

whose context-free skeleton is given by the rules

$$(1) \ \mathrm{A} \ \rightarrow \ \varepsilon \qquad (2) \ \mathrm{A} \ \rightarrow \ \mathrm{A}$$

where the presence of cycles is clear. On the other hand, one look at the DCG is sufficient to detect the refutation of non-cyclic infinite structures of the form

$$\mathrm{a(nil)}, \ \mathrm{a(f(Y_1, nil))}, \ \ldots, \ \mathrm{a(f(Y_{n+1}, f(Y_n, \overset{n}{\ldots} f(Y_1, nil) \overset{n}{\ldots})))}, \ \ldots$$

4.1 Searching for a Condition to Loop

From the previous discussion, we apply 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 studied in [13]. 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 in the context-free backbone of our running example. 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. In this figure, boxes represent the recognition of a grammar category in a given state of the LALR(1) driver.

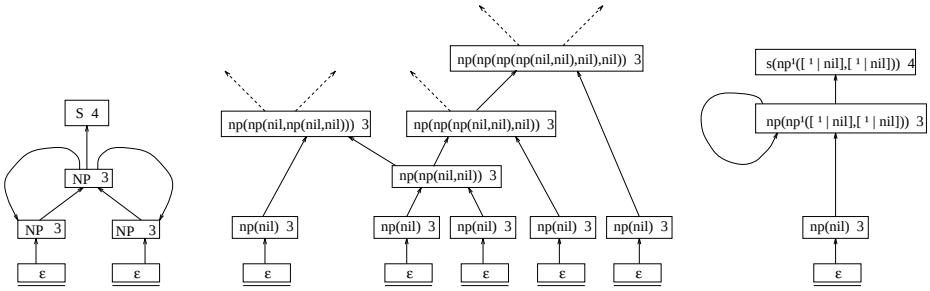

Fig. 7. Cycles in the context-free skeleton and within terms.

To verify now that we can extend cycle detection to predicate symbols, we shall test for unification of the terms implicated. In particular, we must generalize unification to detect cyclic terms with functional symbols.

4.2 Extending Unification and Subsumption

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

Looking for Cycles. After testing the compatibility of name and arity between two terms, 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 cycles, 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 cycles arise 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.

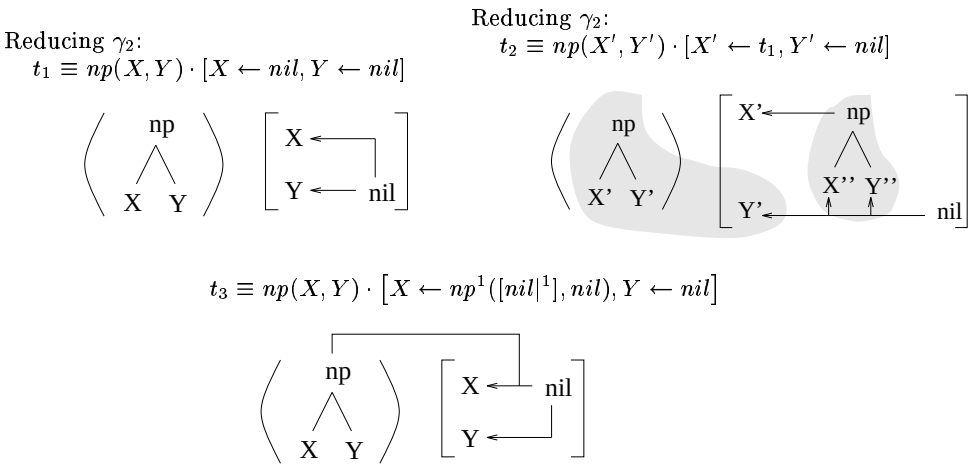

Fig. 8. Cyclic tree traversing (1)

Returning to Fig. 7, once the context-free cycle 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

$$np(nil), np(np(nil,nil)), np(np(np(nil,nil),nil)), \ldots, np(np^1([nil|^1],nil))$$

is generated. In an analogous form, the family

$$np(nil), np(np(nil,nil)), np(np(nil,np(nil,nil))), \ldots, np(np^1(nil,[nil|^1]))$$

can be generated. Due to the sharing of computations the second family is generated from the result of the first derivation, so, by means of the successive applications of clauses γ_2 and γ_4, we will in fact generate the term on the right-hand side of the figure, $np(np^1([nil|^1],[nil|^1]))$, which corresponds exactly to the internal representation of the term[1]. We shall 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 with $np(X)$ and $np(Y)$ in γ_2, and $np(np(X,Y))$ is instantiated, yielding $np(np(nil,nil))$. In the following

[1] it must be pointed out that we could collapse structures $np(nil)$ and $np(np^1([nil|^1],[nil|^1]))$ from the right-hand side of Fig. 7 in $np(^1[nil|np(^{1,1})])$, but this would require a non-trivial additional treatment.

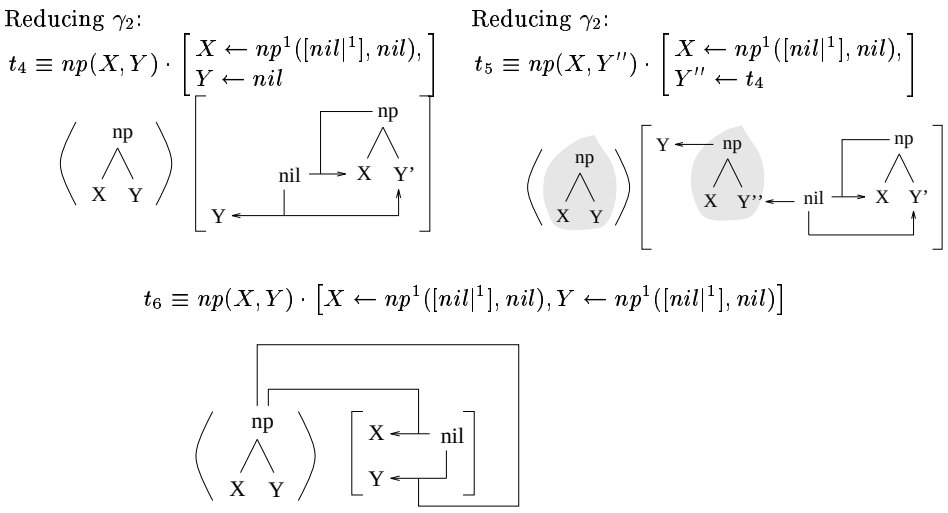

Fig. 9. Cyclic tree traversing (2)

stage, the same step will be performed over $np(np(nil, nil))$ and $np(nil)$, yielding $np(np(np(nil, nil), nil))$. At this point, we consider that:

- there exists a cycle in the context-free backbone,
- we have repeated the same kind of unification twice, and
- the latter has been applied over the result of the former.

Therefore this process can be repeated an unbounded number of times to give terms with the form $np(np^1([nil|^1], nil))$. The same reasoning can be applied if we wish to unify 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.

The unification is explained in detail in Fig. 8. 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. So we have to compare the structures of the arguments associated to predicate symbol np, and in order to clarify the exposition, we have written them as term·substitution. The second term, t_2, is obtained after applying a unification step over the first one, t_1. To show that this step is the same we applied when building t_1, they are shadowed. Now, t_1 and t_2 satisfy the conditions we have established to detect a cycle, namely a cycle exists in the context-free backbone, and we have repeated the same kind of unification twice, the latter over the result of the former. t_3 is the resulting cyclic term. In Fig. 9 we show an analogous operation over t_3.

Cyclic Subsumption and Unification. Now, we will see some examples of how the presence of cyclic terms affects the unification and subsumption operations.

In general, a function subsumes ($\preceq$) another function if it has the same functor and arity and its arguments either are equal or subsume the other function's arguments. When dealing with cyclic terms, one or more arguments can be built from an alternative: another term, or cycling back to the function. Such an argument will subsume another one if it is subsumed by at least one alternative.

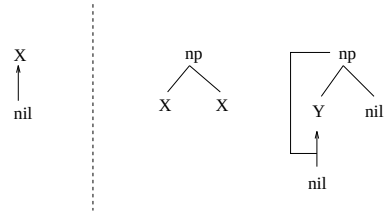

Fig. 10. mgu of substitutions involving cyclic terms.

Returning to the example of Fig. 8, we can conclude that $np^1([nil|^1],^1)$ subsumes $np^1([nil|^1], nil)$. Functor and arity, $np/2$ are the same, and so are the first arguments, $[nil|^1]$, and for the second ones, $[nil|^1] \preceq nil$ because of the first alternative, clearly $nil \preceq nil$.

On the other hand, when calculating the mgu we also have to consider each alternative in the cyclic term, but discarding those that do not match. Thus:

$$\text{mgu}(\{Y \leftarrow [a|b]\}, \{Y \leftarrow a\}) = \{Y \leftarrow a\}$$

and therefore, following the latter example:

$$\text{mgu}(np(X, X), np^1([nil|^1], nil)) = \{X \leftarrow \text{nil}\}$$

which is graphically shown in Fig. 10. Finally, we must not forget that variables are the most general terms and so they subsume any term, even alternatives in cyclic terms. For example:

$$\text{mgu}(np(X), np^1([a|^1])) = \{X \leftarrow [a|np^1([a|^1])\}$$

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, but without omitting information about sub-categorization. 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.

The strategy described does not couple the design of descriptive and operational formalisms [12], nor even limit them [11]. 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 practical constraint is the consideration of monotonous DCGs, that we justify for their practical linguistic interest. This allows us to conceive their use in a grammar development context.

In comparison with algorithms based on the temporary replacement of pointers in structures [4], 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 sharing of structures.

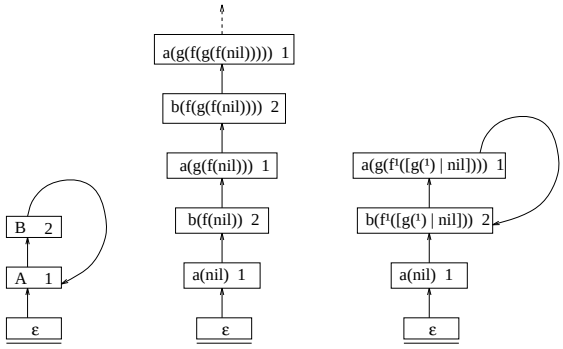

Fig. 11. Cycles in a conjunctive context

Focusing now our attention on methods extending the concept of unification to composed terms [3], the overload for non-cyclic structures is often great. In our case, we minimize this cost factor by a previous filtering phase to detect cycles in the context-free backbone. In the same way, the treatment of monotonous programs in a bottom-up evaluation scheme simplifies the unification protocol.

Finally, we can make reference to algorithms based on the memoization of nodes and comparison to new ones [7]. 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.

6 The Domain of Application

This question has a practical sense since, as has already been established [8], DCGs are only semi-decidable when functional symbols are present and, in consequence, any evaluation strategy dealing with cycles is at stake. To simplify our explanation we shall work in the frame of monotonic first-order logic. We shall prove that our proposal is capable of detecting and traversing cycles with a regular syntactic structure. If this is not the case, the strategy does not guarantee

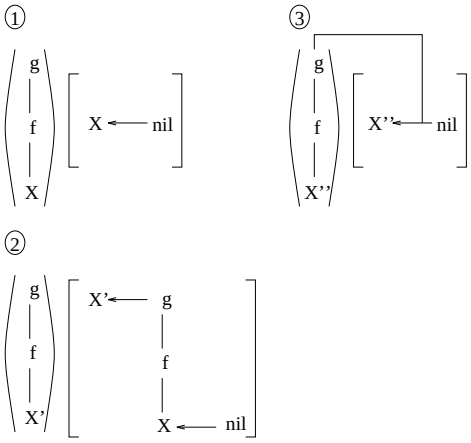

Fig. 12. Cycle traversing in a conjunctive context

termination, which in practical terms is equivalent to saying that the algorithm is equivalent to classic evaluation schema in the worst case.

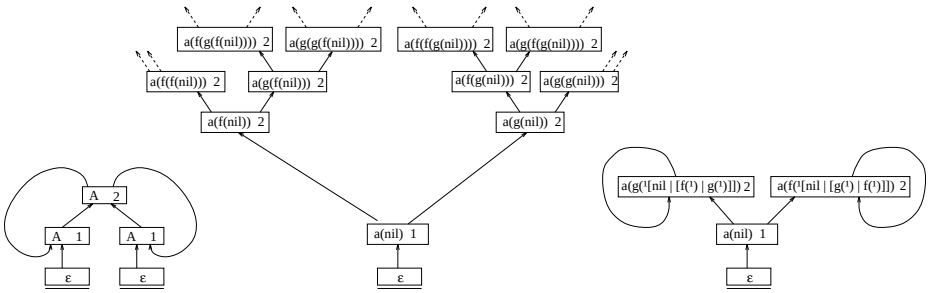

Fig. 13. Cycles in a disjunctive context

Although the technique described could be extended to more complex cycles, our interest in regular ones is justified by the difficulty of representing useful information about non-regular structures in an interactive programming environment, which would reduce the interest of this study to a theoretical one.

To facilitate understanding, and given that technical points have been studied in much greater detail before, we shall informally describe the behavior of our proposal over representative examples covering all possible cases in cycles with a regular structure.

6.1 Conjunctive Terms

This is the simplest case, when all terms included in the cycle are generated from the refutation of clauses without common bodies. This is, for example, the case of the following DCG:

$$\gamma_1: \ \mathrm{a(nil).} \qquad \gamma_2: \ \mathrm{a(g(X))} \ \rightarrow \ \mathrm{b(X).} \qquad \gamma_3: \ \mathrm{b(f(X))} \ \rightarrow \ \mathrm{a(X).}$$

Here, we have a cycle such as is shown in Fig. 11, one for predicate a and another for predicate b; both with depth two. Cycle traversing is illustrated in Fig. 12 for predicate a.

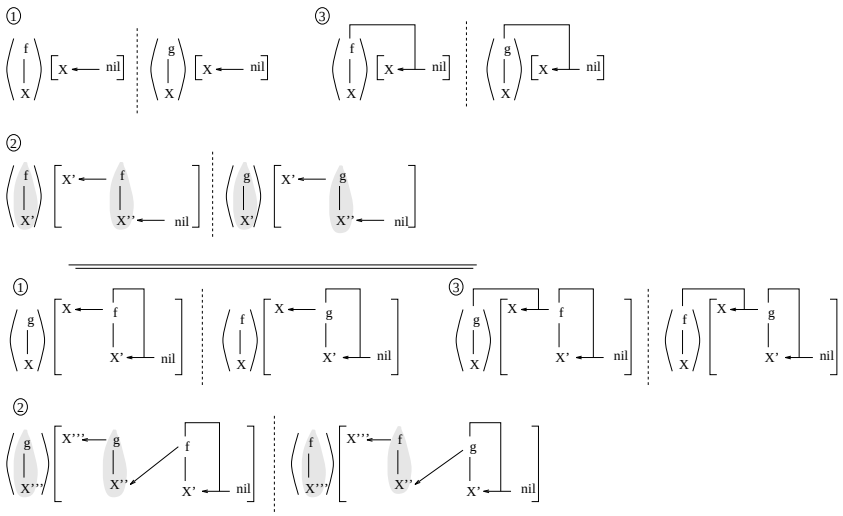

Fig. 14. Cycle traversing in a disjunctive context

6.2 Disjunctive Terms

In this case, we consider that there exist terms in the cycle that have been generated from the refutation of clauses with common bodies. A simple example is given by the DCG defined by the following clauses:

$$\gamma_1: \ \mathrm{a(nil).} \qquad \gamma_2: \ \mathrm{a(f(X))} \ \rightarrow \ \mathrm{a(X).} \qquad \gamma_3: \ \mathrm{a(g(X))} \ \rightarrow \ \mathrm{a(X).}$$

which presents two cycles on predicate a with a disjunction on functions f and g, as is shown in Fig. 13. The cycle traversing is succinctly described in Fig. 14, due to lack of space, only formely steps are shown.

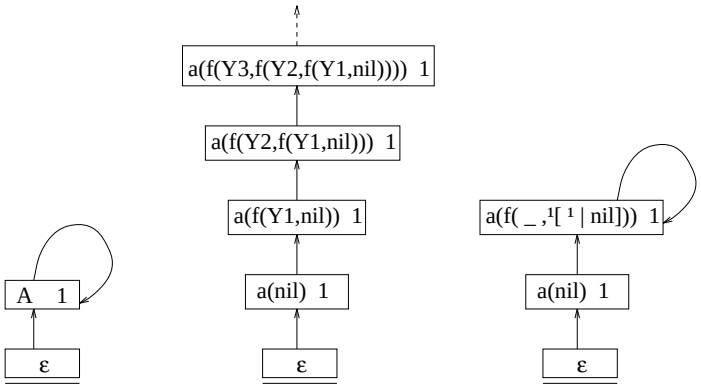

Fig. 15. An infinite structure

6.3 Non-Cyclic Infinite Structures

Finally, our proposal allows us sometimes, but not always, to detect and traverse non-cyclic infinite structures. This is, typically, the case of the presence of anonymous variables in the clauses. Here, we re-take the DCG defined by the clauses:

$$\gamma_1: \ \mathrm{a(nil)}. \qquad \gamma_2: \ \mathrm{a(f(Y,X))} \ \rightarrow \ \mathrm{a(X)}.$$

in which each time the second rule is refuted, the variable Y takes a new value. As a consequence, it is possible to enter the evaluation in an infinite loop in order to generate all possible answers to the request of the type $\rightarrow$ `a(X)`, as is shown in Fig. 15. However, once these anonymous variables have been located[2], it is possible to detect and traverse this kind of structures as we can see in Fig. 16.

7 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:

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

We cannot really provide a comparison with other DCG parsers because of their problems in dealing with cyclic structures. We can however consider results on S^T as a reference for non-dynamic SLR(1)-like methods [6, 10], and naïve dynamic bottom-up methods [5, 2] can be assimilated to S^1 results without synchronization. This information is compiled in Figs. 18 and 17. The former

[2] a simple static study of the DCG is sufficient.

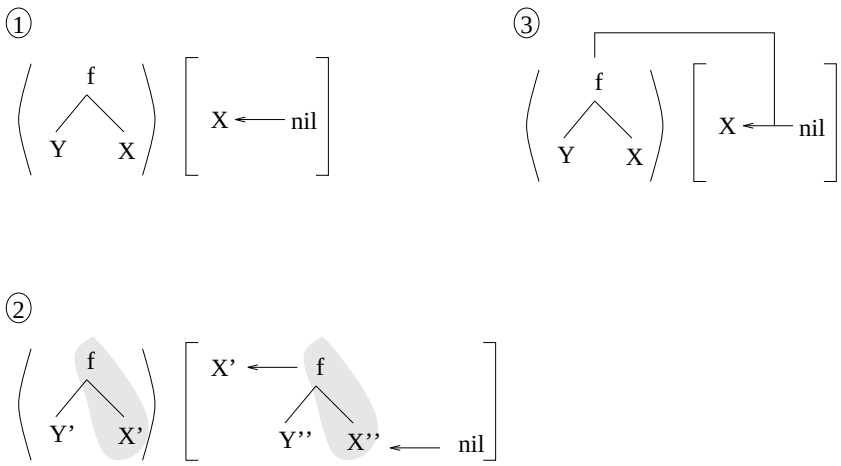

Fig. 16. Traversing a non-cyclic infinite structure

compares the generated items in S^1, S^2 and S^T, the actual number of dynamic transitions generated in S^1 and the original number to be considered if no optimization is applied. The latter 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 to detect cycles.

8 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 DCGs. In this context, we exploit the context-free backbone of these logic programs to efficiently guide detection of regular cyclic constructions without overload for non-cyclic ones.

9 Acknowledgments

This work has been partially supported by projects XUGA 10505B96 and XUGA 20402B97 of the Autonomous Government of Galicia (Xunta de Galicia), project HF97-223 by the Government of Spain, and project 1FD97-0047-C04-02 by the European Community.

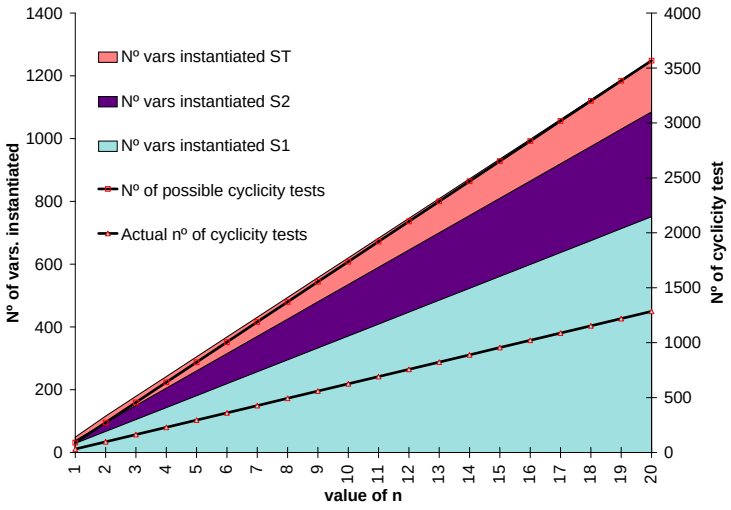

Fig. 17. Some experimental results

References

1. Bresnan, J., Kaplan, T.: Lexical-Functional Grammar: a formal system for grammatical representation. In J. Bresnan (ed.): the Mental Representation of Grammatical Relations (1982) 173–281. MIT Press
2. De la Clergerie, E.: Automates à piles et programmation dynamique. DyALog: une application à la Programmation en Logique. PhD thesis. University of Paris 7 (1993)
3. Filgueiras, M.: A PROLOG interpreter working with infinite terms. In *Implementations of* PROLOG (1985)
4. Haridi, S., Sahlin, D.:. Efficient implementation of unification of cyclic structures. In *Implementations of* PROLOG (1985)
5. Lang, B.: Towards a uniform formal framework for parsing. In M. Tomita (ed.): Current Issues in Parsing Technology (1991) 153–171. Kluwer Academic Publishers
6. Nilsson, U.: AID: an alternative implementation of DCGs. New Generation Computing **4** (1986) 383–399
7. Nilsson, M., Tanaka, H.: Cyclic tree traversal. Lecture Notes in Computer Science **225** (1986) 593–599. Springer-Verlag
8. Pereira, F.C.N., Warren, D.H.D.: Parsing as Deduction. Proc. of the 21^{st} Annual Metting of the Association for Computational Linguistics (1983) 137-144. Cambridge, Mass.
9. Prawitz, D.: Natural Deduction, Proof-Theoretical Study (1965). Almqvist & Wiksell, Stockholm, Sweden
10. Rosenblueth, D. A., Peralta, J. C.: LR inference: inference systems for fixed-mode logic programs, based on LR parsing. International Logic Programming Symposium (1994) 439–453. The MIT Press
11. Shieber, S. M.: Using restriction to extend parsing algorithms for complex-feature-based formalisms. 23th Annual Meeting of the ACL (1985) 145–152

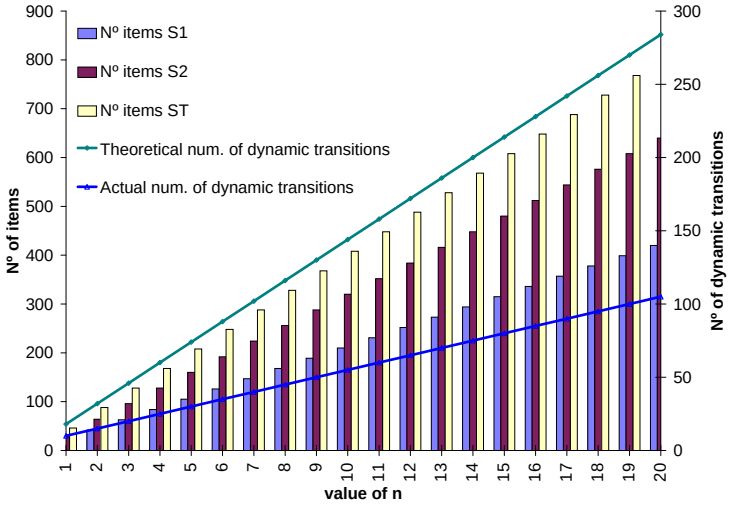

Fig. 18. Some experimental results

12. Stolzenburg, F.: Membership-constraints and some applications. Technical Report Fachberichte Informatik **5/94** (1994). Universität Koblenz-Landau, Koblenz
13. Vilares, M.: Efficient Incremental Parsing for Context-Free Languages. PhD thesis. University of Nice (1992)
14. Vilares, M., Alonso, M. A.: An LALR extension for DCGs in dynamic programming. In C. Martín Vide (ed.): Mathematical Linguistics II (1997). John Benjamins Publishing Company