On Non-Termination in DCGs

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

[1] Department of Informatics
University of Vigo,
Campus As Lagoas, s/n
32004 Orense, Spain
http://grupocole.org
[2] Department of Computer Science
University of A Coruña
Faculty of Informatics, Campus de Elviña s/n
15071 A Coruña, Spain
e-mail: {vilares,cabrero,alonso}@udc.es

Abstract. Our goal is to study a practical approach to deal with non-termination in definite clause grammars. We focus on two problems, loop and cyclic structure detection and representation, maintaining a tight balance between practical efficiency and operational completeness. In order to guarantee the validity of our conclusions, we first map our study to a common framework, where the effectiveness of each approach will be examined and, later, compared by running experiments.

1 Introduction

Non-termination is a crucial problem when encoding definite clause grammars (DCGs). By termination we mean here finiteness of all possible logical derivations starting from the initial goal. We can claim this for decidable problems, although in practice we are often confronted with the problem that an apparently correct program may fail to terminate for certain forms of the input. It is, typically, the case of PROLOG programs with left-recursion on local variables. This difference between theoretical and practical operational models is justified by efficiency gains, assuming that this kind of situations can be usually avoided in practical applications by alert programmers. However, the descriptive potential offered by unrestricted declarative programming is appreciated in language development tasks, where a large completion domain allows the modelling effort to be saved.

Previous works on this subject often focus on strategies for proving termination in letf-terminating programs. This is the case of Apt and Pedresdi in [2], or Ullman and Van Elder in [6]. However, these studies are limited to deal with left-recursion in top-down resolution and do not provide a practical approach to represent infinite derivations. A different point of view is given by Filgueiras in [3], providing effective representation for cyclic structures.

We focus on two problems that arise when working with DCGs, both of which can cause non-termination. The first problem is of general interest in formal

grammar theory and it concerns loop detection, when the parsing process is repeatedly returned to the same processing state. The second problem stems from cyclic structures and it is a known consequence of non implementing the occur-check, which would forbid unification of a variable with a term in which it occurs. In both cases, we rely on strategies to represent cyclic derivations and structures.

Our proposal takes place in the framework of resolution strategies based on dynamic programming, not-limited to top-down approaches, while extending the concept of unification to composed terms. Although the key idea of dynamic programming is to keep traces of computations to achieve computation sharing, it also offers flexibility to investigate loop detection. To deal with cyclic structures, our approach refines the occur-check to minimize the time spent checking for re-occurring variables.

2 A situated framework

We consider as parsing frame the logical push-down automaton (LPDA) [7]. 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*. The *stack* of the automaton is a finite sequence of *items* $[A, it, bp, st].\sigma$, where the top is on the left, $A \in T_\Delta[\mathcal{F} \cup \mathcal{X}]$, σ is a substitution, it is the current input position, st is a state for a driver controlling the evaluation; and bp is the back-pointer, the position in this input string at which we began to look for that configuration of the LPDA. Dynamic programming is introduced by collapsing stack representations on a fixed number of items and adapting transitions in order to deal with these items. When the correctness and completeness of computations are assured, we talk about the concept of *dynamic frame*. Here, the use of it allows us to index the parse, which relies on the concept of *itemset*, associating a set of items to each token in the input string. Transitions in a dynamic frame are of three kinds:

- *Horizontal:* $B \mapsto C\{A\}$. 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:* $BD \mapsto C\{A\}$. 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:* $B \mapsto CB\{A\}$. We can apply it to stacks $E.\rho\ \xi$, iff there is $\sigma = \text{mgu}(E, B)$, such that $F\sigma = A\sigma$, for F a fact in the extensional database. We obtain the stack $C\sigma.\sigma\ B.\rho\ \xi$.

where B, C and D are items and $A \in T_\Sigma[\mathcal{F} \cup \mathcal{X}]$, a control condition to operate the transition. Two dynamic frames are of practical interest, S^2 and S^1, where the superscript denotes the number of top stack elements used to generate items.

The standard dynamic frame, S^T, where a stack is given by all its components, uses backtracking to simulate non-determinism.

2.1 Cyclic structures

Most interpreters do not implement the occur-check, this making it possible to unify a variable with a term in which it occurs, producing an infinite term. To prevent this, we extend substitutions to function and predicate symbols [3].

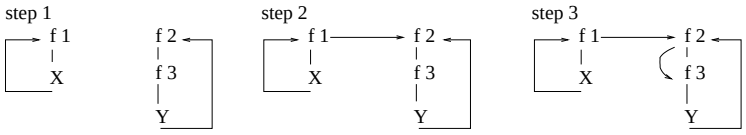

Fig. 1. Unification of X with Y.

To illustrate our discussion we consider terms resulting from solving `unify(X,f(X))` and `unify(Y,f(f(Y)))`, whose unification is shown in Fig. 1, where we shall use "$\rightarrow$" to denote a unification link from a represented symbol to its representative. The actions to be performed start by dereferencing X and Y. The process leads to the unification of $f1$ with $f2$, symbols with the same name and arity. A link is established from $f1$ to $f2$, as is shown in step 2 of Fig. 1. We now proceed with the unification of the arguments X and $f3$. After dereferencing X to $f1$ and then to $f2$, it results in the unification of $f2$ and $f3$. Then a link is added, as is shown in step 3 of Fig. 1. Finally, we have to unify $f3$ and Y, which is dereferenced to $f2$ and then to $f3$, and it is equal to $f3$.

2.2 Loop detection

Loop detection resorts to noticing when the parser is repeatedly returned to the same state. As an example, consider the following naïve grammar:

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

By starting with `b`, you will expect the parser to find that $X \rightarrow f^1([\text{nil}|^1])$. To achieve this, we can construct the following sequence of terms:

$$a(\text{nil}), a(f(\text{nil})), a(f(f(\text{nil}))), \ldots$$

If we just use the former algorithm to check the subsumption of two terms like $f(\text{nil})$ and $f(f(\text{nil}))$, it fails as shown in Fig.2. The loop is never detected and the analysis process lasts forever. To create any such answer, we have to resort to cyclic derivations [5]. We recover the context-free backbone, the context-free grammar obtained by removing all the arguments from the predicates of the DCG. Any cyclic derivation over a DCG will have a corresponding cyclic derivation over

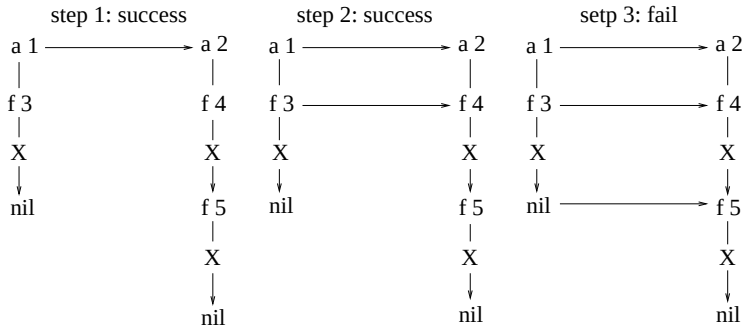

Fig. 2. Trying to subsume $f(\text{nil})$ and $f(f(\text{nil}))$.

this skeleton. Once a context-free loop is detected, we traverse for predicate and function symbols to detect whether the analysis has returned to a previous state. To achieve this, we store the terms in a shared structure that allows us to easily detect whether a term occurs inside another one. In our example, the context-free backbone is

$$r_1 : a \to b. \qquad r_2 : a \to a.$$

and, after generating the terms $a(f(\text{nil})), a(f(f(\text{nil})))$ we detect a context-free loop, $a \equiv a$. We now traverse the terms as shown in Fig. 3, concluding that the first one occurs inside the latter one, returning to the same processing state and a loop has been completed, and we have detected it.

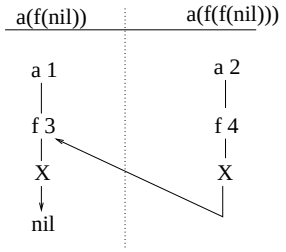

Fig. 3. Traversing $f(\text{nil})$ and $f(f(\text{nil}))$ after context-free loop detection.

3 The evaluation schema

It is possible to efficiently guide the detection of cyclic derivations on the basis of the evaluation strategy used. On the other hand, for cycles to arise in arguments, it is first necessary for the context-free backbone given by the predicate symbols to determine the recognition of a same syntactic category without extra work

for the scanning mode. Our aim is to estimate which evaluation scheme is the most appropriate to deal with this problem.

We consider three schema on a frame S^1: pure bottom-up, a mixed-strategy with dynamic prediction [4], and a mixed-strategy with static prediction [7]. In order to formalize these, we introduce:

- The vector $\boldsymbol{T}_k$ of the variables occurring in each rule $\gamma_k : A_{k,0} \rightarrow A_{k,1}, \ldots, A_{k,n_k}$.
- The predicate symbols $\bigtriangledown_{k,i}$, $i \in \{1, \ldots, n_k\}$.

The meaning of an instance of $\bigtriangledown_{k,i}(\boldsymbol{T}_k)$ will be dependent on the parsing scheme.

3.1 A mixed-strategy with dynamic prediction

Here, the symbol $\bigtriangledown_{k,i}$ shows that the first i categories in the right-hand-side of rule γ_k have already been recognized. In addition, given a category $A_{k,i}$, we shall consider the associated symbols $A'_{k,i}$ and $A''_{k,i}$ to respectively indicate that $A_{k,i}$ is yet to be recognized or has been already recognized. We obtain the following set of transitions that characterize the parsing strategy:

1. $[\$, 0, 0, \_] \mapsto [A'_{0,0}, 0, 0, \_]\$$
2. $[A'_{k,0}, it, it, \_] \mapsto [\bigtriangledown_{k,0}(\boldsymbol{T}_k), it, it, \_]$ $[A'_{k,0}, it, it, \_]$
3. $[\bigtriangledown_{k,i}(\boldsymbol{T}_k), it, bp, \_] \mapsto [A'_{k,i+1}, it, it, \_]$ $[\bigtriangledown_{k,i}(\boldsymbol{T}_k), it, bp, \_]$
4. $[\bigtriangledown_{k,n_k}(\boldsymbol{T}_k), it, bp, \_]$ $[A'_{k,0}, bp, bp, \_] \mapsto [A''_{k,0}, it, bp, \_]$
5. $[A''_{k,i+1}, it, bp, \_]$ $[\bigtriangledown_{k,i}(\boldsymbol{T}_k), bp, r, \_] \mapsto [\bigtriangledown_{k,i+1}(\boldsymbol{T}_k), it, r, \_]$

that we informally explain as:

1. States the axiom $A_{0,0}$, which we represent by $A'_{0,0}$.
2. Requires the recognition of $A'_{k,0}$, which we represent by $\bigtriangledown_{k,0}$.
3. Selects the leftmost unrecognized category, $A'_{k,i+1}$.
4. The body of γ_k has been recognized. We push $A''_{k,0}$ to show that $A'_{k,0}$ has been recognized.
5. After recognition of $A''_{k,i+1}$, the parse advances to the next term in γ_k.

where the state, represented by "_", has no operative sense here.

3.2 A bottom-up scheme

Here, the symbol $\bigtriangledown_{k,i}$ expresses that the categories in the right-hand-side of γ_k after the i position have already been recognized. The set of transitions is:

1. $M \mapsto \bigtriangledown_{k,n_k}(\boldsymbol{T}_k)\ M$
2. $\bigtriangledown_{k,i}(\boldsymbol{T}_k)\ A_{k,i} \mapsto \bigtriangledown_{k,i-1}(\boldsymbol{T}_k)$
3. $\bigtriangledown_{k,0}(\boldsymbol{T}_k) \mapsto A_{k,0}$

M is defined by giving a vector of new variables of appropriate length as argument to every predicate of the LPDA. We interpret the transitions as follows:

1. Ignoring the literal M on top of the stack, select an arbitrary clause γ_k whose head is to be proved; then push the position literal $\bigtriangledown_{k,n_k}$ on the stack to indicate that none of the body literal has yet been proved.
2. We reduce the stack and increment the position literal from $\bigtriangledown_{k,i}$ to $\bigtriangledown_{k,i-1}$.
3. 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 clause, since it is proved.

An obvious improvement to this construction is to select only new clause instances whose head may help prove a negative literal of the current clause.

3.3 A mixed-strategy with static prediction

We require the same interpretation for symbols $\bigtriangledown_{k,i}$ as for bottom-up evaluation:

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

where *action*, *goto*, *shift* and *reduce* are well-known concepts in LR parsing [1]. We interpret these transitions as follows:

1. Pushes $\bigtriangledown_{k,n_k}$ to indicate that the body of γ_k is to be recognized.
2. The parser advances after the refutation of $\mathrm{A}_{k,i}$.
3. All literals in the body of γ_k have been recognized, and $A_{k,0}$ is recognized.
4. Pushes the literal $A_{k,i+1}$, assuming that it will be needed for the recognition.
5. Begins with the recognition of γ_k.
6. States the axiom $A_{0,0}$.

Control conditions are built from actions in a driver given by an LALR(1) automaton built from the context-free skeleton.

4 Dealing with cyclic derivations

In order to introduce both LPDA interpretation and cyclic derivations [8], we consider as a running example a simple DCG to deal with the sequentialization of nouns in English, as in the case of *"North Atlantic Treaty Organization"*:

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

The augmented context-free skeleton is now given by the context-free rules:

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

whose characteristic finite state machine is shown in Fig. 4. The recognition of the complete sentence ends with a reduction by clause γ_1, obtaining the term $s(np(np(\text{"}North\text{"},\text{"}Atlantic\text{"})))$ 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 that the processing of this branch has finished. The resulting configurations are depicted in Fig. 5.

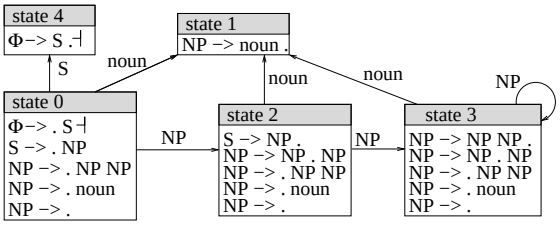

Fig. 4. Characteristic finite state machine for the running example

However, the grammar actually 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, as is shown in Fig. 6. Here, the left-most drawing represents the cycle in the context-free backbone, the following one the parsing process on the DCG in state 3, and the last a finite description for the infinite term traversal. Boxes represent the recognition of a grammar category in a given state of the LALR(1) driver.

4.1 Looking for loops

After testing the compatibility of name and arity between two terms in different items, the algorithm establishes whether 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. This is equivalent to testing the existence of a loop for these non-terminals in the context-free backbone.

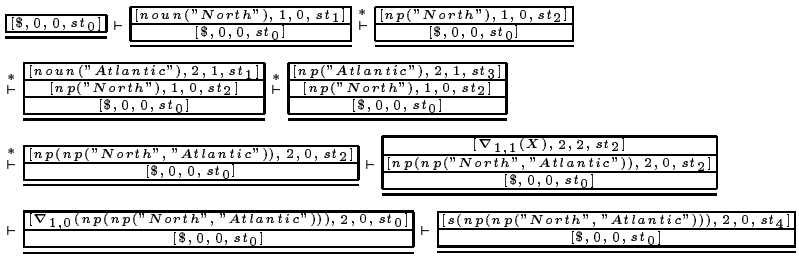

Fig. 5. Configurations for recognizing the sentence "*North Atlantic*".

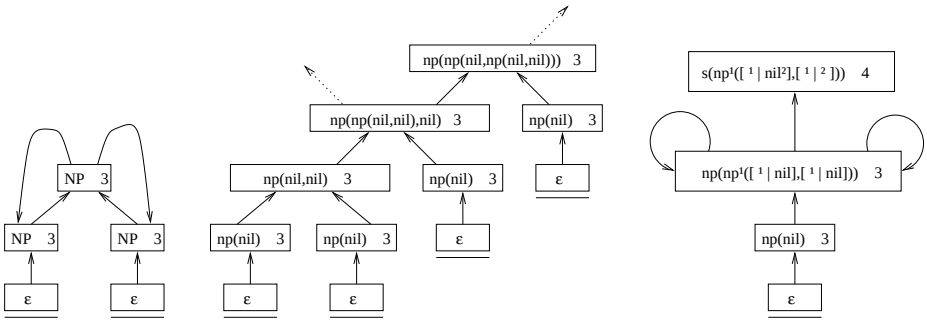

Fig. 6. Cycles in the context-free skeleton and within terms.

If all these comparisons succeed, we look for loops. The system 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 common sub-structures exist if and only if any of these comparisons succeed. In this last case, the algorithm stops on the pair of arguments concerned, while continuing with the rest of the arguments. Returning to Fig. 6, once the context-free loop has been detected, we check for possible cyclic derivation in the original DCG. The center drawing in that figure shows how the family of terms

$$\mathrm{np(nil)}, \mathrm{np(np(nil,nil))}, \mathrm{np(np(np(nil,nil),nil))}, \ldots, \mathrm{np(np^1([nil—^1],nil))}$$

is generated. In an analogous form, the family

$$\mathrm{np(nil)}, \mathrm{np(np(nil,nil))}, \mathrm{np(np(nil,np(nil,nil)))}, \ldots, \mathrm{np(np^1(nil,[nil—^1]))}$$

can be also 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 shall in fact generate the term on the right-hand side of the figure, $np(np^1([nil^2|^1],[^2|^1]))$, 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 $np(X,Y)$ is instantiated, yielding $np(np(nil,nil))$. In the following 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 derivation 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.6 shows the compact representation we use in this case. 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.

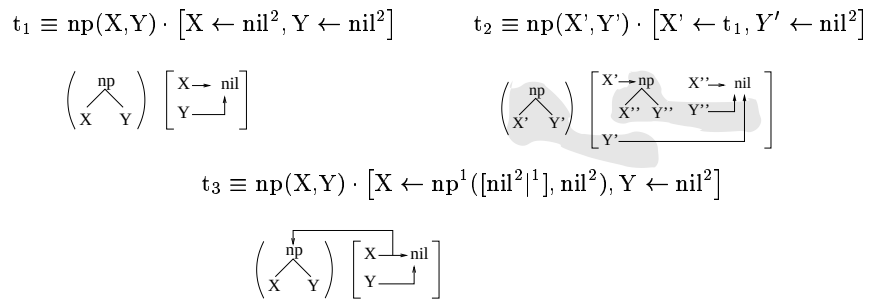

Fig. 7. Cyclic tree traversing (1)

Loop detection is explained in detail in Fig. 7. The terms to be studied 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 as the one we applied when building t_1, they are shadowed. Now, t_1 and t_2 satisfy the conditions we have established to detect a loop, namely a loop exists in the context-free backbone, and we have repeated the same kind of derivation twice, the latter over the result of the former. Thus, t_3 is the resulting loop representation.

[1] We could collapse structures $np(nil)$ and $np(np^1([nil^2|^1],[^2|^1]))$ from Fig. 6 in $np(^1[nil|np(^{1,1})])$, but this would require a non-trivial additional treatment.

4.2 Cyclic subsumption and unification

Now, we shall see some examples of how the presence of cyclic structures 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 are either equal or subsume the other function's arguments. When dealing with cyclic structures, 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.

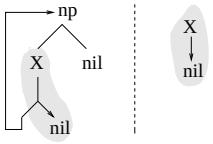

Fig. 8. mgu of substitutions involving cyclic terms.

Returning to the example of Fig. 7, 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. 8. For better understanding, the matching parts of substitutions are shadowed. 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 Experimental results

Our running grammar contains a rule NP $\rightarrow$ NP NP. So, the number of cyclic parses grows exponentially with the length, n, of the phrase. This number is:

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

In this context, we are not interested in traversing cyclic structures [8] since the technique considered in our framework is not dependent on the parsing scheme used. At this point, efficiency is only a consequence of the capacity of the evaluation strategy to filter out useless items. We focus on loop detection, comparing performances over the schema previously introduced. We assume that lexical information is directly provided by a tagger since only syntactic phenomena are of interest for us. In this manner, Fig. 9 shows the number of

items compared in order to detect cyclic derivations. These experiments have been performed on S^1, the optimal dynamic frame in each case [9].

So, we can observe the efficiency of mixed-strategies including static prediction in contrast with pure bottom-up approaches or evaluators based on dynamic prediction. This confirms the real interest of using a driver as guideline to deal with cyclic derivations, as contrasted with naïve subsumption-based strategies.

6 Conclusion

We have described two problems which can cause non-termination in DCG parsing. The first problem involves the ability of the parser for loop detection and representation. We have tackled the question from the viewpoint of dynamic programming, exploiting domain ordering, improving tabular evaluation, and profiting from the analogy with classic context-free parsing.

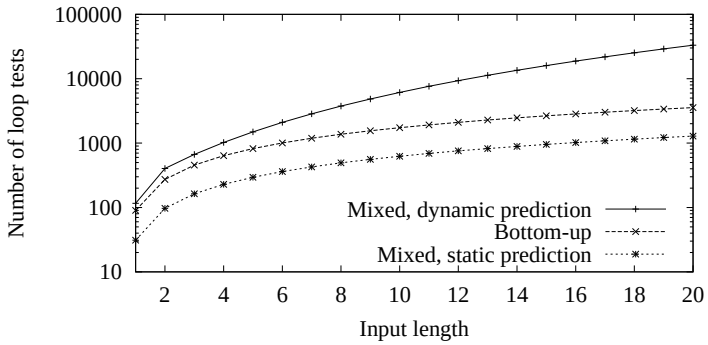

Fig. 9. Number of tests for loop detection with different parsing schema

The second problem is to detect and represent cyclic structures in finite time. Our proposal generalizes the concept of unification to include function and predicate symbol substitution, making use of the sharing properties in dynamic programming evaluation in order to reduce the computational complexity.

Acknowledgments

This work has been partially supported by the Spanish Government under projects TIC2000-0370-C02-01 and HP2001-0044, and the Autonomous Government of Galicia under project PGIDT01PXI10506PN.

Significantly revised and enlarged version of a paper published in A. Gelbukh (Ed.). CICLing-2000: Computational Linguistics and Intelligent Text Processing, CIC-IPN, Mexico City, 2000.

References

1. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1-2. Prentice-Hall, Englewood Cliff, New Jersey, U.S.A., 1973.
2. K.R. Apt and D. Pedreschi. Studies in pure PROLOG: Termination. In J.W. Lloyd, editor, *Computational Logic*, volume 1436 of *Basic Research Series*, pages 150–176. Springer-Verlag, Berlin-Heidelberg-New York, 1990.
3. M. Filgueiras. A PROLOG interpreter working with infinite terms. *Implementations of* PROLOG, 1984.
4. F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proc. of the* 21^{st} *Annual Metting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, U.S.A., 1983.
5. C. Samuelss. Avoiding non-termination in unification grammars. In *Proc. of the 4th Int. Workshop on Natural Language Understanding and Logic Programming*, pages 4–16, Nara, Japan, 1993.
6. J.D. Ullman and A. van Gelder. Efficient tests for top-down termination of logical rules. *Journal of ACM*, 2(35):345–373, 1988.
7. M. Vilares and M.A. Alonso. An LALR extension for DCGs in dynamic programming. In Carlos Martín Vide, editor, *Mathematical and Computational Analysis of Natural Language*, volume 45 of *Studies in Functional and Structural Linguistics*, pages 267–278. John Benjamins Publishing Company, Amsterdam & Philadelphia, 1998.
8. M. Vilares, M.A. Alonso, and D. Cabrero. An operational model for parsing definite clause grammars with infinite terms. In Alain Lecomte, François Lamarche, and Guy Perrier, editors, *Logical Aspects of Computational Linguistics*, volume 1582 of *Lecture Notes in Artificial Intelligence*, pages 212–230. Springer-Verlag, Berlin-Heidelberg-New York, 1999.
9. M. Vilares, D. Cabrero, and M. A. Alonso. Dynamic programming as frame for efficient parsing. In *18th Int. Conf. of SCCC*, pages 68–75, Piscataway, NJ, 1998. IEEE Press.