On Non-Termination in DCGs

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

The objective of this paper 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 situated 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 unification-based grammar formalisms, although practical systems often diverge from their theoretical definitions. By termination we mean here finiteness of all possible logical derivations starting in the initial goal. Although from a theoretical point of view, we can claim it for decidable problems, often logical problems are confronted with the problem that an apparently correct program may fail to terminate for certain forms of the input. This 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 modeling effort to be saved.

Previous works on this subject often focus on strategies for proving termination in left-terminating programs. This is the case of [Apt & Pedresdi 1990] or [Ullman & Van Elder 1988]. 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 [Filguerias 1984], providing effective representation for cyclic structures.

We focus on two problems that arise when working with definite clause grammars (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 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

As first goal, we structure our work within a well-defined frame. This will allows us to compare different approaches on the basis of an common descriptive formalism, focusing on strategy-dependent features.

2.1 The parsing model

We consider a uniform parsing frame, the logical push-down automaton (LPDA), such as it is introduced in [Vilares & Alonso 1998].

An LPDA is defined as 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 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 a driver controlling the evaluation. Transitions are of three kinds:

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

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

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

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. We use bp to chain pop transitions.

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.

2.2 Cyclic structures

Conventional interpreters do not implement the occur-check in the unification algorithm. Doing so, it is possible to unify a variable with a term in which it occurs, producing an infinite circular term.

To prevent this, we chose to work in the generalization of substitution to function and predicate symbols, as initially proposed in [Filgueiras 1984]. This means that the unification algorithm will treat symbols in the same way as for variables: referencing or linking them whenever they unify, and dereferencing before testing for compatibility.

To illustrate our discussion we consider, as a classic example, terms resulting from solving `unify(X,f(X))` and `unify(Y,f(f(Y)))`, as shown in step 1 of Fig. 1. When using a conventional unification algorithm, without occur-check, it will loop trying to unify X with Y.

Following now [Filgueiras 1984], the process is also shown in Fig. 1, where we shall use $\rightarrow$ to denote a unification link from a represented symbol to its representative. Here, the actions to be performed start by dereferencing X and Y. The unification process leads to the unification of $f1$ with $f2$, since both symbols

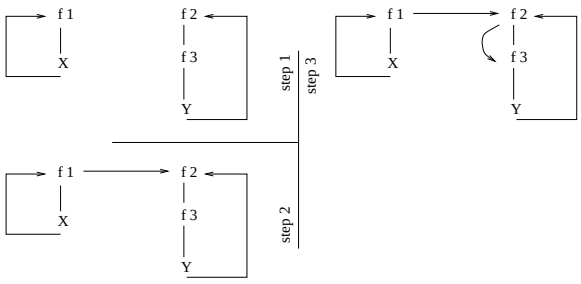


Figure 1: Unification of X with Y.

have the same name and arity. So, a link is set from $f1$ to $f2$, as it 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 $f3$. As consequence, a new link is added, as it 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 $f3$. The algorithm stops here, unifying X and Y.

2.3 Loop detection

Loop detection resorts to noticing when the process is repeatedly returned to the same processing state. In context-free parsing, the comparison of one state to a previous one involves checking for equality between atomic symbols. However, DCGs can be thought of a generalization of non-terminal symbols from a finite domain of atomic elements to a possibly infinite domain of directed graph structures and, thus, the equality test is insufficient. Instead, we have to compare terms using subsumption. As an example, considerer the following naïve grammar:

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

By starting with the atom b, you will expect the analysis process 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 [Samuels 1993].

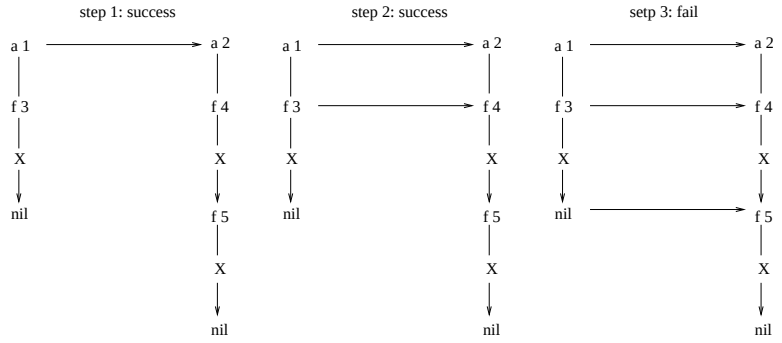


Figure 2: Trying to subsume $f(nil)$ and $f(f(nil))$.

From a practical viewpoint, given a DCG, we recover its context-free backbone, that is, the context-free grammar obtained by removing all the arguments from the predicates of the grammar. It is obvious that any cyclic derivation over a DCG will have a corresponding cyclic derivation over this skeleton. So, before checking for a loop in the DCG itself, we shall check the context-free backbone. Once a loop is detected, we traverse for predicate and function symbols to detect whether the analysis has returned to a previous state.

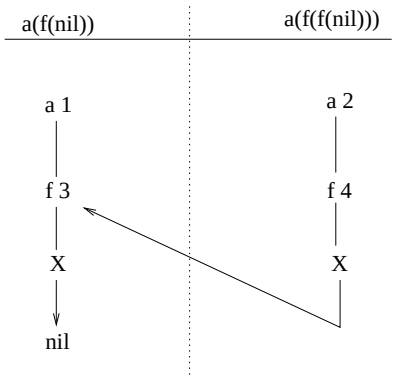


Figure 3: Traversing $f(nil)$ and $f(f(nil))$ after context-free loop detection.

In order to achieve this, we store the terms in a shared structure that allows us to easily detect whether a term occurs inside another one, and, therefore, they are the beginning and the end of a loop iteration in the analysis process. So, in the previous example, the context-free backbone is

$$r_1: \quad a \rightarrow b. \qquad\qquad r_2: \quad a \rightarrow a.$$

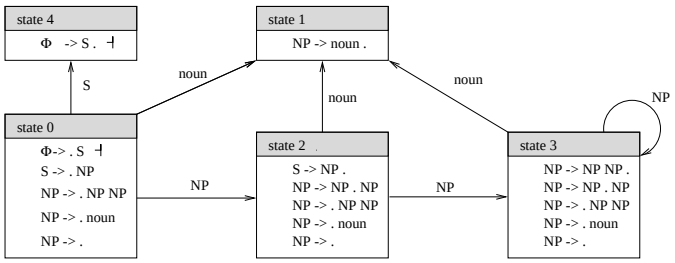


Figure 4: Characteristic finite state machine for the running example

and, after generating the terms

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

we detect a context-free loop, $a \equiv a$. We traverse the terms as shown in Fig. 3, concluding that the first one occurs inside the latter one, returning to the same processing state[1] and, thus, a loop has been completed, and we have detected it.

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 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. This is a key observation to solve infinite term traversal, and our aim is to estimate which evaluation scheme is the most appropriated to deal with.

We have considered three basic evaluation schema: a pure bottom-up architecture, a mixed-strategy with dynamic prediction [Pereira & Warren 1983], and a mixed-strategy with static prediction [Vilares & Alonso 1998]. In this manner, we can compare the computational behaviour over a family of bottom-up related evaluators working on a same dynamic frame S^1.

To locate each scheme in our framework, we introduce the categories $\bigtriangledown_{k,i},\ i \in \{1, \ldots, n_k\}$ for each rule $\gamma_k : A_{k,0} \rightarrow A_{k,1}, \ldots, A_{k,n_k}$, whose meaning will be dependent on the parsing scheme. For further details, the reader can see [Vilares, Cabrero & Alonso 1998].

[1] We didn't go further in the analysis of the input, and the reduced non-terminal is the same.

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. So, we obtain the following set of transitions that characterize the parsing strategy:

$$
\begin{array}{lll}
1. & [\$, 0, 0, \_] & \longmapsto \quad [A'_{0,0}, 0, 0, \_]\ \$ \\
2. & [A'_{k,0}, it, it, \_] & \longmapsto \quad [\bigtriangledown_{k,0}(\vec{T_k}), it, it, \_] \\
 & & \qquad\quad [A'_{k,0}, it, it, \_] \\
3. & [\bigtriangledown_{k,i}(\vec{T_k}), it, bp, \_] & \longmapsto \quad [A'_{k,i+1}, it, it, \_] \\
 & & \qquad\quad [\bigtriangledown_{k,i}(\vec{T_k}), it, bp, \_] \\
4. & [\bigtriangledown_{k,n_k}(\vec{T_k}), it, bp, \_] & \\
 & [A'_{k,0}, bp, bp, \_] & \longmapsto \quad [A''_{k,0}, it, bp, \_] \\
5. & [A''_{k,i+1}, it, bp, \_] & \\
 & [\bigtriangledown_{k,i}(\vec{T_k}), bp, r, \_] & \longmapsto \quad [\bigtriangledown_{k,i+1}(\vec{T_k}), it, r, \_]
\end{array}
$$

where an instance of $\bigtriangledown_{k,i}(\vec{T_k})$ indicates that all literals until the i^{th} literal in the body of γ_k have been proved. 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:

$$
\begin{array}{llll}
1. & M & \longmapsto \quad \bigtriangledown_{k,n_k}(\vec{T_k}) & M \\
2. & \bigtriangledown_{k,i}(\vec{T_k}) \quad A_{k,i} & \longmapsto \quad \bigtriangledown_{k,i-1}(\vec{T_k}) & \\
3. & \bigtriangledown_{k,0}(\vec{T_k}) & \longmapsto \quad A_{k,0} &
\end{array}
$$

where M is an atom defined by giving as argument to every predicate of the LPDA a vector of new variables of appropriate length, and $\bigtriangledown_{k,i}(\vec{T_k})$ indicates that all literals from the i^{th} in the body of the clause γ_k, have been proved.

3.3 A mixed-strategy with static prediction

We requires the same interpretation for symbols $\bigtriangledown_{k,i}$ as for bottom-up evaluation. We define the transitions as follows:

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

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

3.4 Parsing a sample sentence

To introduce both, LPDA interpretation and cyclic derivations, 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"*. 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). $\qquad$ γ_2 : np(np(X, Y)) $\rightarrow$ np(X) np(Y).
γ_3 : np(X) $\rightarrow$ noun(X). $\qquad$ γ_4 : np(nil).

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

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

whose characteristic finite state machine is shown in Fig. 4.

We are going to describe the parsing process for the simple sentence "*North Atlantic*", focusing on the introduced mixed-strategy with static prediction. 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 $[noun("North"), 0, 1, st_1]$ that indicates the recognition of term $noun("North")$ 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. 5.

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

Figure 5: 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. 6.

$$\vdash \boxed{\begin{array}{c}[\nabla_{3,1}(X), 1, 1, st_1]\\ [noun("North"), 1, 0, st_1]\\ [\$, 0, 0, st_0]\end{array}} \vdash \boxed{\begin{array}{c}[\nabla_{3,0}("North"), 1, 0, st_0]\\ [\$, 0, 0, st_0]\end{array}} \vdash \boxed{\begin{array}{c}[np("North"), 1, 0, st_2]\\ [\$, 0, 0, st_0]\end{array}}$$

Figure 6: Configuration during the reduction of clause γ_3.

We can now scan the word "*Atlantic*", resulting in the recognizing of the term $noun("Atlantic")$ 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. 7.

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

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

Figure 7: Configurations during the processing of the word "Atlantic".

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

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

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. 9.

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, such it is shown in Fig. 10. Here, the left-most drawing represents the cycle in the context-free backbone, the following 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.

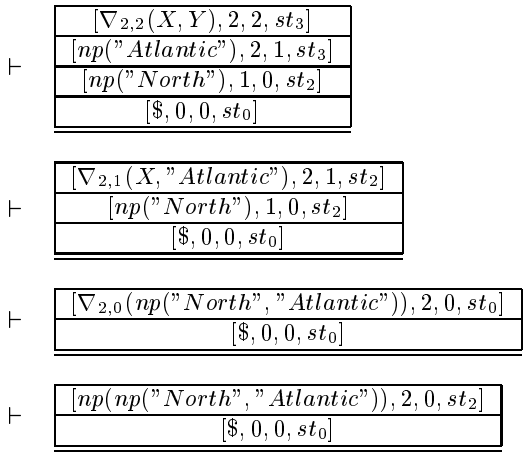


Figure 8: Recognition of the nominal phrase "North Atlantic".

4 DEALING WITH CYCLIC DERIVATIONS

We can now explore with greater depth into the adaptation of the general loop and cyclic detection strategies introduced in our situated framework to the set of parsing schema considered in the dynamic frame S^1. To facilitate the understanding, we shall focus on our running example, assuming the adaptation to the other schema in a natural manner.

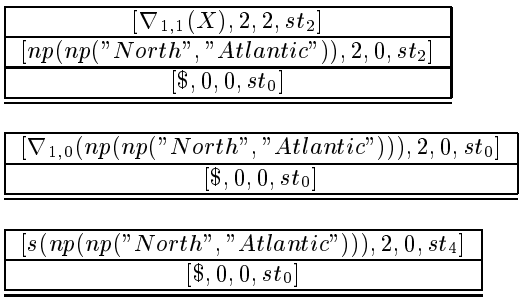


*Figure 9: Configurations for the recognizing of the sentence "*North Atlantic*".*

4.1 Looking for loops

After testing the compatibility of name and arity between two terms in different items, the algorithm establishes if the associated non-terminals in the driver have been generated in the same state[2], covering the same portion of the text, which is equivalent to comparing the corresponding back-pointers. This is equivalent to test the existence of a loop for these non-terminals in the context-free backbone.

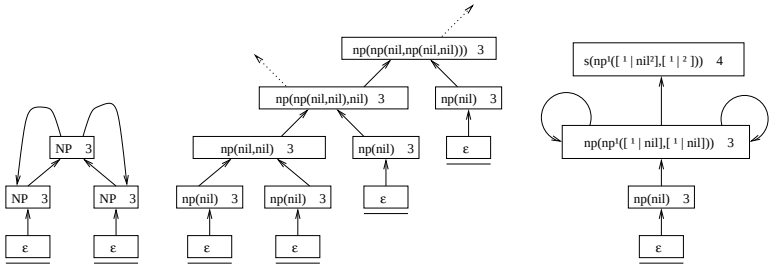


Figure 10: Cycles in the context-free skeleton and within terms.

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 there exists common sub-structures 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.

Once the context-free loop has been detected, we check for cyclic derivations in the original DCG. The center drawing in Fig. 10 shows how the family of terms

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

[2]This would be only necessary in the mixed-strategy with static prediction, because for the other schema states have no operative sense.

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 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[3]. 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.

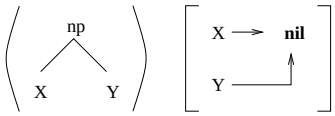
Therefore this process can be repeated an unbounded number of times to give terms with the form $np(np^1([nil|^1], nil))$. The same reasoning can be applied if we wish to unify with the variable Y. The right-hand drawing in Fig.10 shows the compact representation we use in this case. The functor np is 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.

Loop detection is explained in detail in Fig. 11. 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 that 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.

[3]We could collapse structures $np(nil)$ and $np(np^1([nil^2|^1], [^2|^1]))$ from the right-hand side of Fig. 10 in $np(^1[nil|np(^1,^1)])$, but this would require a non-trivial additional treatment.

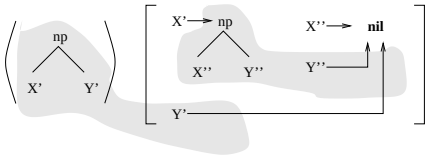
Reducing γ_2:

$t_1 \equiv np(X,Y)\cdot[X \leftarrow nil^2, Y \leftarrow nil^2]$



Reducing γ_2:

$t_2 \equiv np(X',Y')\cdot[X' \leftarrow t_1, Y' \leftarrow nil^2]$



$$t_3 \equiv \quad np(X,Y)\cdot[X \leftarrow np^1([nil^2|^1], nil^2), Y \leftarrow nil^2]$$

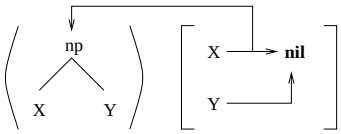


Figure 11: Cyclic tree traversing (1)

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 either are 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.

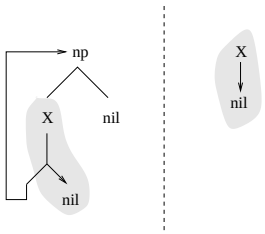


Figure 12: mgu of substitutions involving cyclic terms.

Returning to the example of Fig. 11, 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. 12. 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

For the tests we take our running example dealing with sequentialization of nouns. 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:

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

We are not here interested in time and space bounds related to traversing cyclic structures [Vilares, Alonso & Cabrero 1999] since the technique considered in our situated 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. So, we focus now on loop detection, comparing performances over the schema previously introduced.

We assume that lexical information is directly provided by a specialized tagger since only syntactic phenomena are of interest for us. In this manner, Fig. 13 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 [Vilares, Cabrero & Alonso 1998].

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

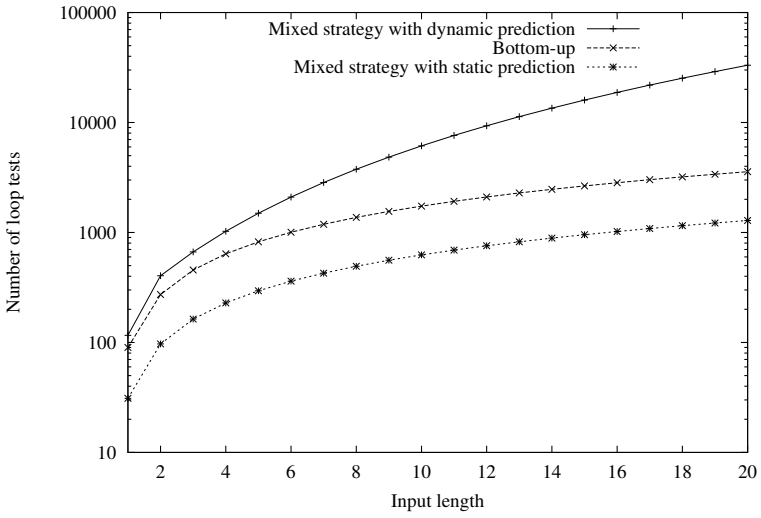


Figure 13: Number of tests for loop detection with different parsing schema

6 CONCLUSIONS

We have discussed and described some possible solutions to two common problems which can cause non-termination in DCG parsing.

The first problem involves the ability of the parser for loop detection and representation. Here, we have tackled the question from the viewpoint of dynamic programming, exploiting the domain ordering, improving tabular evaluation, and profiting from the analogy with classic context-free parsing.

The second problem is to detect and represent cyclic structures in finite time. This is more of a logic programming question, where often available algorithms are related to strategies for traversing cyclic lists. In this case, 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.

REFERENCES

K.R. Apt and D. Pedreschi, 1990. *Studies in pure PROLOG: Termination*. In Computational Logic, vol. 1436 of Basic Research Series, pages 150-176. Springer-Verlag, Berlin-Heidelberg-New York.

M. Filgueiras, 1983. *A PROLOG interpreter working with infinite terms*. Implementations of PROLOG.

F.C.N. Pereira and D.H.D. Warren, 1983. *Parsing as deduction*. In Proc. of the 21^{th} Annual Meeting of the Association for Computational Linguistics, pages 137-144. Cambridge, Massachusetts, U.S.A.

C. Samuels, 1993. *Avoiding non-termination in unification grammars*. In Proc. of 4^{th} Int. Workshop on Natural Language Understanding and Logic Programming, pages 4-16. Nara, Japan.

J.D. Ullman and A. van Gelder. 1988. *Efficient tests for top-down termination of logical rules*. Journal of ACM, 2(35), pages 345-373.

M. Vilares, D. Cabrero and M.A. Alonso, 1998. *Dynamic programming as a frame for efficient parsing*. In 18^{th} Int. Conference of SCCC, IEEE Press., Piscataway, NJ.

M. Vilares and M.A. Alonso, 1998. *An LALR extension for DCGs in dynamic programming*. In. Mathematical and Computational Analysis of Natural Language, vol. 45 of Studies in Functional and Structural Linguistics, pages 267-278. John Benjamin's Publishing Company, Amsterdam & Philadelphia.

M. Vilares, M.A. Alonso and D. Cabrero, 1999. *An operational model for parsing definite clause grammars with infinite terms*. In Logical Aspects of Computational Linguistics, vol. 1582 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin-Heidelberg.New-York.

Manuel Vilares Ferro is a professor at the Computer Science Department, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, SPAIN. *vilares@dc.fi.udc.es*.

David Cabrero Souto is a doctoral student at the Computer Science Department, University of A Coruña. *cabrero@dc.fi.udc.es*.

Miguel A. Alonso Pardo is an associated professor at the Computer Science Department, University of A Coruña. *alonso@dc.fi.udc.es*.

This work has been partially supported by projects XUGA 20402B97 and PGIDT99XI10502B of the Autonomous Government of Galicia, and project 1FD97-0047-C04-02 by the European Community.