Datalog Grammars

Veronica Dahl
Logic Programming Group and
Computing Sciences Department
Simon Fraser University

Paul Tarau
Département d'Informatique
Université de Moncton
and Logic Programming Group
Simon Fraser University

Yan-Nong Huang
Logic Programming Group and
Computing Sciences Department
Simon Fraser University

Abstract

Datalog Grammars (DLGs) are a variant of Definite Clause Grammars that do not use function symbols. For every given input phrase and DCG-grammar we construct a Datalog program and a query which succeeds if and only if the input sequence is recognized by the original grammar.

Finiteness of the Herbrand Universe, under appropriate execution strategies, ensures termination of Datalog grammars. A reverse mapping from Datalog to CF-grammars shows that in principle they have the same computational power on given inputs (queries) of bounded length.

Datalog grammars are more efficient than their DCG counterparts under (terminating) OLDT-resolution because the absence of compound terms induces a smaller tabulation overhead.

A typical cross-fertilization from the field of deductive databases is the use of incremental updates. We show how the use of incremental techniques can be of benefit for the efficient evaluation of Datalog grammars in a typical situation where the input sequence is partially unknown and has to be guessed from the syntactical and morphological context.

However, some of the advanced uses of DCGs that assume a list representation for the connect/3 relation, cannot be recovered by Datalog representation. We give some examples and give sufficient conditions for the equivalence of the two translations.

Keywords: Logic grammars, Deductive Databases, Compilation of DCG-grammars to Datalog, CF-grammars, OLDT-resolution, Evaluation of Datalog programs

1 Introduction

Logic grammars, introduced in 1975 [4], add flexibility and expressive power to logic programming languages, by providing a rewriting paradigm for the solution of problems that involve transforming one representation into another. Thus, they have been useful for natural and formal language analysis, synthesis and translation (see e.g. [15, 1, 13, 12, 5]),

as well as for less traditional applications of grammars, such as compiler implementation [4, 16], general problem solving (e.g. [17]) and software specification ([1]).

However, their usual implementation, which uses lists to represent the strings being analysed or synthesized, results in undesirable complexity properties. The presence of the list constructor makes the Herbrand Universe infinite, and properties like termination or polynomial time execution, which held for context-free grammars, can no longer be taken for granted, even in the case of the simplest type of logic grammar: DCGs.

Of course, substantial NL applications also tend to introduce infinite Herbrand universes, by using functors to construct desired representations from the sentences being parsed.

The aim of this paper is to overcome the termination and complexity introduced by the usual list representation, by using a largely equivalent assertional representation, and showing that under appropriate evaluation mechanisms, such as OLDT-resolution, the termination and complexity properties of the original CF-grammar are preserved. This, moreover, can be achieved even in the presence of extra arguments, such as those typically used to build syntactic or semantic representations as a by-product of parsing, or as a starting point in generation.

2 An overview of Logic Grammars and their habitual translation into Prolog

The simplest and most common type of logic grammars, DCGs, consists of context-free type rules whose symbols can include arguments, and which also admit calls to Prolog predicates. Their list-based compilation into Prolog translates any Prolog calls into themselves, and adds two more arguments to each non-terminal symbol nt. These arguments can be thought of in analysis as the input string to be analysed, and the output string to be passed on as input to the next symbol- i.e., what remains of the input string after some of it has been recognized as being of category nt.

For instance, the following rule might be used in a grammar to define a sentence as a noun phrase represented X followed by a verb phrase represented Y and in which X and Y exhibit agreement of some sort as defined by the agree predicate (e.g., same gender and number for the subject X as for the verb in Y):

```
s --> noun_phrase(X), verb-phrase(Y), {agree(X,Y)}.
```

Its well-known translation into Prolog reads:

```
s(P0,P):- noun_phrase(X,P0,P1), verb_phrase(Y,P1,P), agree(X,Y).
```

If s is called for the input argument `P0=[the,winner,hesitate]`, for instance (and assuming appropriate definitions for the rest of the grammar), this rule will unify `P1` with [hesitate] and `P` with [] before failing due to subject-verb disagreement.

DCGs also admit more than one symbol on the left-hand side, provided the non-initial ones are terminals. This can always be forced through temporarily "terminalizing" symbols as necessary, e.g.[2]:

[1] James Hanlon, personal communication, 1993

[2] This rule is not linguistically motivated, but provides a simple example of the power of the multiple-head grammar rules: a relative pronoun followed in the parsing state by a name, verb and noun phrase (these having been temporarily terminalized for convenience) rewrites into an actual pronoun, *that*, and reconstructs its right context as "name,verb", thus deleting the expectation of a noun phrase after the verb.

```
name --> [name].
verb --> [verb].
```

```
(R)  relative_pronoun, [name], [verb], [noun_phrase] --> [that], name, verb.
```

These rules can be used to delete the expectation of an object noun-phrase in a relative clause, as in "The house *that Jack built* _ fell down".

The process of terminalization (originally called "normalization") has been shown to generate an equivalent grammar, in the sense that all strings described by the original grammar will also be described by the transformed grammar [4]. In generation mode, however, it may be possible to wrongly synthesize additional "sentences" which contain the pseudo-terminals introduced by terminalization.

When translating into Prolog, the pseudo-terminals in left-hand sides of rules will find their way into symbol arguments. For instance, rule (R) above translates into:

```
rel_pronoun(A,E):-
    'C'(A,that,B),name(B,C),verb(C,D),
    'C(E,name,F),
    'C'(F,verb,G),
    'C(G,noun_phrase,D).
```

Notice that left-hand side terminals find their way into the output argument of a symbol (i.e., they are synthesized), whereas right-hand side ones will appear in input arguments (i.e., they are analyzed).

The variable E takes the value `[name,verb,noun_phrase|D]`, so that this Prolog rule is equivalent to the rule:

```
rel_pronoun([that|B],[noun,verb,noun_phrase|D]):- name(B,C), verb(C,D).
```

More general types of logic grammars have been devised to provide further expressive power, but they shall not occupy us greatly here. These have been covered in [1].

3 Datalog Grammars

Datalog grammars (DLG) are a variant of DCG-grammars which use a `'connect'` predicate without function symbols and a set of facts described by an assertional representation of the input string.

3.1 DCG grammars and their limitations

The reasons why we consider DLGs interesting as an alternative to standard DCGs are related to simpler semantics, better computational behaviour and more efficient implementation. They address the following deficiencies of DCGs, some of which originate on their list-based implementation, some of which are inherited from Prolog:

- infinite Herbrand Universe
- non-termination
- unnecessary recomputation
- structure creation on the heap
- lists (a sequential) data structure are bottleneck for multithreaded execution

3.2 An assertional representation: back to the future

Since the beginning of Logic Programming as a distinct paradigm one of the founders of the discipline, R.A. Kowalski has pointed out in Logic and Problem Solving [9] that either an assertional representations or data-structure representation can be used for the same purpose in logic programs.

The DCG-formalism itself leaves open the implementation of the connect/3 predicate linking the database of terminals with the rule part of the grammar. However, in the past list-based representations have dominated logic grammars[3].

Note that a connect/3 predicate like

```
'C'(From,Word,[Word|To]).
```

can be replaced by

```
'D'(N1,Word,N2):-N2 is N1+1.
```

We can also compute N2 at compile time and have therefore only pure DATALOG facts like for instance

```
'D'(12,happy,13).
```

For first-argument indexing purposes this becomes:

```
'D'(happy,12,13).
```

Alternatively we could have introduced a new predicate for each known word i.e. something like

```
happy(S1,S2):-succ(S1,S2).
```

for succ/2 defined as a pure DATALOG predicate of the form

```
succ(0,1).
succ(1,2).
....
succ(N1,N).
```

where N is the length of the input sequence and N1 is `N-1`, while keeping around

```
'D'(Word,From,To):-w1(From,To).
....
'D'(Word,From,To):-wK(From,To).
```

for each word in the dictionary to deal with variables (unknown words) in the input sequence.

We will not talk about this representation here, as it can be considered simply as an optimization that 'partially evaluates' the grammar one step further with respect to the known input sequence.

We believe that a pure Datalog version of DCG grammars is useful in the context of Logic Programming Languages for the following reasons:

- no sequential constraints on the order of execution (giving for instance better OR-parallel execution)

[3] One reason is convenient manipulation through unification. Another, less obvious, comes from the fact that in sequential implementations of Prolog there's no penalty for using inherently sequential data structures like lists. Moreover, Prolog was always considered more practical than Deductive Database systems although recent developments show that this may change in the future.

- free movement to left or to right by using the (reversible) successor relation instead of the implicit unidirectional list pointers inaccessible at Prolog level[4]
- possibility to use deductive database techniques for their evaluation
- Termination is inherited form Datalog programs
- Tracing and debugging becomes easier because of explicit word ordering

3.3 A DLG Preprocessor for Prolog

A DLG-preprocessor can be obtained easily by modifying a standard DCG-preprocessor.

Remark that we can simply replace the list oriented 'connect' predicate by the automatically generated set of DATALOG facts describing the input sequence. We have actually done this by modifying R.A. O'Keefe's public domain DCG-preprocessor. The program is available by e-mail upon request from tarau@cs.sfu.ca.

For the input grammar:

```
sentence-->np, vp.

np-->art,adjs,n.
np-->art,n.

adjs-->adjs,adj.
adjs-->adj.

vp-->v.

n-->[elephant].

art-->[the].

adj-->[green].
adj-->[greedy].
adj-->[little].

v-->[flies].

'-->'([the,little,green,elephant,flies]).
```

we obtain the following DATALOG program:

```
sentence(A,B) :-
        np(A,C),
        vp(C,B).

np(A,B) :-
        art(A,C),
        adjs(C,D),
        n(D,B).


adjs(A,B) :-
        adjs(A,C),
        adj(C,B).
adjs(A,B) :-
       adj(A,B).

vp(A,B) :-  v(A,B).

?- sentence(0,5).

n(A,B) :- 'D'(elephant,A,B).
art(A,B) :- 'D'(the,A,B).
adj(A,B) :- 'D'(little,A,B).
adj(A,B) :- 'D'(green,A,B).
adj(A,B) :- 'D'(greedy,A,B).
v(A,B) :-  'D'(flies,A,B).

'D'(the,0,1).
'D'(little,1,2).
'D'(green,2,3).
'D'(elephant,3,4).
'D'(flies,4,5).
```

We require the user to name the grammar's start symbol as "sentence", so that the preprocessor can reconstruct the appropriate query (e.g., `sentence(0,5)`) from its Datalog representation.

[4] This is useful in error recovery.

3.4 Datalog programs and CF-grammars

Theorem 1 *Let G be a CF-grammar with "sentence" as its initial symbol and I an input to be recognized by G. Then there's a DATALOG program* `DL(G,I)` *and a goal* `DL(A,I)` *which succeeds if and only if I is recognized by G.*

Proof. The proof is immediate, through the following construction.

Definition 1 *Given a context-free grammar G and an input sentence I, we define a Datalog program DL(G, I) as follows.*

1. *The input sentence* $I = \omega_1\omega_2 \ldots \omega_m$ *originates the clauses* $'D'(\omega_1, 0, 1)$, $'D'(\omega_2, 1, 2)$, $\ldots, 'D'(\omega_m, m-1, m)$.
2. *a lexical rule* $P \rightarrow word$ *originates the clause:* $P(A, B) \leftarrow' D'(word, A, B)$.
3. *a phrase-structure rule of the form* $P \rightarrow P_1, P_2, \ldots, P_n$ *originates the clause* $P(A_0, A_1) : -P'_1, P'_2, \ldots, P'_n$., *where*

$$\begin{cases} P'_i = P_i(A_{i-1}, A_i) & if P_i \text{ is a non-terminal symbol} \\ P'_i = 'D'(p_i, A_{i-1}, A_i) & if P_i \text{ is a terminal symbol} \end{cases}$$

Theorem 2 *For every DATALOG program and ground query there's a context-free grammar and an input phrase derived from the query such that the query is answered affirmatively by the program if and only if the phrase is recognized by the grammar.*

Proof. For each ground instance $a_0(x_0^1, \ldots x_0^k) : -\ldots a_i(x_i^1 \ldots x_i^l), \ldots$ of a Datalog clause we define the rule $a_0 x_0^1, \ldots x_0^k \rightarrow \ldots a_i x_i^1 \ldots x_i^l \ldots$ where $a_i x_i^1 \ldots x_i^l$ is a new symbol obtained by concatenating in an obvious way the syntactic components of $a_i(x_i^1 \ldots x_i^l)$. Then every ground instance of a clause is uniquely represented by a grammar rule. It is easy to show by induction that this mapping extends to a mapping from Datalog derivations to grammar derivations for instance in the case of the T_P operator that evaluates the Datalog program bottom-up. A similar operation is applied to every ground instance of the query.

These theorems relate in a straightforward way two formalisms known to have common properties and evaluation techniques[5].

3.5 Adding non-Datalog arguments to Datalog grammars

First, let us remark that some linguistic constraints (like gender and person) can be easily expressed by using only Datalog arguments.

However, DCG-grammars have been useful especially when extra arguments were added to express various constraints to avoid overgeneration and to build structures resulting from the parsing process. Artificial restrictions on the nature of the arguments would seem unpleasant for the users of logic grammars.

What happens therefore when we add to DLG-grammar some extra 'non-Datalog' arguments (i.e. arguments containing function symbols that make the Herbrand Universe infinite)?

The following definitions and theorem give an obvious answer to this question.

[5] Bottom-up evaluation techniques and memoing have been present for a long time both in the study of formal languages and deductive databases, but we do not know about a study comparing them explicitly.

Definition 2 *Let D be a DLG-grammar and P a definite program obtained by adding extra arguments to D and then using Definition 1. Let T_D be the proof-tree obtained for a terminating Datalog evaluation of D. Let T_P be the proof tree obtained from T_D by keeping only the derivations which still succeed after adding argument unification at each step. Let G be a (Datalog) goal for D and G' a goal for P obtained by instantiating to some arguments of G arbitrary (not necessarily Datalog) terms. A* DLG-answer *to P is a substitution θ obtained by composing the substitutions of a successful derivation of T_P.*

Theorem 3 *The set of DLG-answers is finite and every DLG-answer is a computed answer of P. The algorithm to compute all the DLG-answers of a program always terminates.*

`Proof.` It follows immediately from the fact that extra arguments subject to possibly failing (but always terminating) unifications in T_P can only fail sooner than their Datalog counterparts in T_D.

Clearly any decidable constraint-solving mechanism can be used here instead of unification.

Note also that writing and testing a grammar (which happens to be a DLG) and then adding extra arguments is current programming practice. By considering only DLG-answers one may loose completeness although the intuitively relevant set of answers will still be found.

Additional arguments are often orthogonal to the syntactic derivation process described by the grammar and they either act as constraints to limit an otherwise overgenerating grammar or to build semantic representations in a compositional style.

4 Using DLG grammars with incomplete information

An interesting application for DLG grammars is the case where the input string has some unknown words. This may arise for instance when using a DLG grammar processor to provide feed-back to a speech recognition system that failed to recognize part of the input.

The termination properties of DLG grammars are very important in this case. As we will show in the next section, incremental techniques for deductive database updates can be used in this case to speed-up the search for a missing word. This is a typical case where DLG grammars make real use of a key advantage of logic grammars: the ability to do both generation and recognition.

Another typical application is a misspelled word in a programming language or natural language interface to a database system which can be forgiven after checking the alternatives with the user.

Consider our sample grammar with the incomplete sentence to be parsed:

```
The little __ elephant ___
```

(this can be a real world problem for speech recognition e. g. in the presence of noise for a machine or a hearing impairment for a human). Our grammar completes the missing words with plausible ones (i.e., words with the appropriate syntactic category) from the vocabulary, simply upon the statement of the query:

```
'--> ([the,little,X,elephant,Y]).
```

More dramatically, in some examples the choices made for the missing words may result in sentences with different structures. For instance, the incomplete sentence

```
'--> ([the,blue,X,Y]).
```

can be completed in two ways: "The blue sky shines", in which the first three words constitute the subject noun phrase, and "blue" is an adjective denoting a colour, and "The blue sounds sad", in which the subject has only two words, and "blue" is a noun denoting a type of music.

In naive examples such as the above, and imagining a very large vocabulary, there will in general be many ways of filling in the missing words. Presumably, more phonetic, syntactic and semantic information will also be available in practice (e.g. more nuanced syntactic categories, types, the number of syllables, and possibly an incomplete or partially correct version of the missing word).

Of course, incomplete input can also be parsed in DCGs, but with no guarantee of termination.

5 Applying Incremental Deductive Database Techniques

Recall that a finite number of Horn clauses (or rules) without function symbol constitutes a Datalog program (or deductive database) where rules with empty bodies form the extensional database (EDB) and the remaining rules the intensional database (IDB). A predicate defined by the EDB is called the 'base predicate' whereas a predicate defined by the IDB is called the 'derived predicate'. Here only acyclic proofs are considered because one can always guarantee the strict increase in the length of the words recognized (by elimination of empty rules in the initial grammar, for example).

The semantics of a deductive database is determined by its least Herbrand model, and this model can be procedurally obtained. The well-known semi-naive evaluation algorithm [3] is one of such procedures. In this algorithm, we begin with the set of axioms and by applying the derivation rules we obtain the theorems of the first 'layer'; then we take these theorems as new starting point, by the application of derivation rules, to derive the theorems of the second 'layer'; and so on. Generally, to derive the theorems of next 'layer', at least one theorem produced at the previous stage must be used. This process terminates when no more new theorems can be generated.

With respect to previous bottom-up or mixed solutions to the problems of non-termination and recomputation of already derived consequences (such as Earley parsing[6], Earley deduction [14], memoing [18] and magic-set transformations [2]), this approach exploits the specific features of DLG to reduce the bookkeeping overhead present in those methods: subsumption testing is made simpler through hashing[7]. The management of alternative binding environments is also less cumbersome in the incremental approach. An interesting top-down methodology which automates the transformation of left-recursive DGS into non-left recursive ones was recently developed in [6], in view of reversible grammars, and could also be used for our purposes, although we have not studied its comparative efficiency.

5.1 Bottom-Up Generation and Maintenance Algorithms

In order to count the number of different proofs for each theorem, we need to rewrite a given deductive database $DB = EDB \cup IDB$ to $DB^c = EDB^c \cup IDB^c$, the later is augmented with counters. That is,

- If $a(t_1, t_2, \ldots, t_n). \in EDB$, then,
  $a(t_1, t_2, \ldots, t_n, 1). \in EDB^c$.

[6] D.H.D. Warren, unpublished note 1975

[7] Hashing on ground terms (or terms ground up to a given depth) can be done as in most Prologs through the `term_hash/2` primitive.

- If $h(Y) : -a_1(X_1), a_2(X_2), \ldots, a_n(X_n). \in IDB$ where Y, X_i are vector variables and $n > 0$, then,
  $h(Y, c_1 \times c_2 \times \ldots \times c_n) : -a_1(X_1, c_1), a_2(X_2, c_2), \ldots, a_n(X_n, c_n). \in IDB^c$.

Based upon the semi-naive evaluation, the Proof Counting Algorithm is given below. Notice that facts with the counters being equal to 0 are deleted automatically during as well as after the computation.

Proof Counting Algorithm
Input: $DB^c = EDB^c \cup IDB^c$: a deductive database augmented with counters;
Output: S^c: set of theorems;
Method:
Begin
1) $\Delta S^c := EDB^c$;
2) $S^c := \emptyset$;
3) while $\Delta S^c \neq \emptyset$ do
4) $\quad \Delta S^c_{pivot} := diff\_I^c(S^c, \Delta S^c)$;
5) $\quad S^c := S^c \uplus \Delta S^c$;
6) $\quad \Delta S^c := \Delta S^c_{pivot}$;
7) endwhile;
End;

where, the differential immediate consequence operator is defined as follows,

$diff\_I^c(S^c, \Delta S^c) =$
$|\{\{h(Y, c_1 \times c_2 \times \ldots \times c_n)\theta : h(Y, c_1 \times c_2 \times \ldots \times c_n) :$
$-a_1(X_1, c_1), a_2(X_2, c_2), \ldots, a_n(X_n, c_n). \in IDB^c, \theta$ is a substitution such that
$\forall i \in [1, n], a_i(X_i, c_i)\theta \in S^c \cup \Delta S^c$ and $\exists j \in [1, n]$ such that $a_j(X_j, c_j)\theta \in \Delta S^c\}\}|$

$\{\{\}\}$ denotes multi-set. $||$ is the rectifying operation, namely, elements with the identical theorem contents are combined into a single one by adding their counters. The $\uplus$ operation is the ordinary union operation followed by the rectifying operation.

Theorem 4 *Let DB be a logic database; then, for any theorem t, the value of the counter attribute of t yielded by calling the above algorithm is equal to the number of different proofs for t.*

The proof for this theorem is given in [8].

Given a deductive database DB, and the set of theorems derived from it, if the EDB receives a update, how can we obtain the new set of theorems without obviously recomputing from scratch?

If EDB^c is the extensional database augmented with counters, ΔEDB^c is used to represent the update to EDB^c. For elements in ΔEDB^c, the counter value is initialized to 1 for inserted facts, and to -1 for deleted ones.

Recomputing from scratch in general is not an efficient method, because updates might possibly influence a small portion of the set of derived theorems, thus only this part should be reconsidered. To take advantage of already available information, we propose an incremental computation.

Counter Maintenance Algorithm

Input: $DB^c = EDB^c \cup IDB^c$: a logic database augmented with counters;
S^c: old set of theorems; ΔEDB^c: update to EDB^c
Output: S^c: new set of theorems;
Method:
Begin
1) $\Delta S^c := \Delta EDB^c$;
2) while $\Delta S^c \neq \emptyset$ do
3) $\quad \Delta S^c_{pivot} := diff\_I^c(S^c, \Delta S^c)$;
4) $\quad S^c := S^c \uplus \Delta S^c$;
5) $\quad \Delta S^c := \Delta S^c_{pivot}$;
6) endwhile;
End;

Theorem 5 *The counter maintenance algorithm maintains correctly the set of theorems.*

Readers can refer to [8] for the proof of this theorem.

5.2 A Datalog Grammar Example

For the example given in the previous section, the problem of verifying the syntactical correctness w.r.t. to the context-free grammar, of the initial sentence

'the little green elephant flies'

becomes that of checking whether 'axiom(0, 5)' is in the set of theorems of the Datalog program transformed. Thus we may produce the whole set of derivable theorems for the Datalog program in a bottom-up way. It is to be noted that we do not over-compute anything in doing so. In the first place, every base fact in the Datalog program will necessarily be used in deriving our target goal, so other top-down deductive database methods (such as Magic-set [2]) are not likely to be used to further reduce the search scope. Secondly, to produce and materialize the whole set of theorems (together with their proof numbers), the syntactical analyses on two similar sentences can be done in an 'incremental' way.

We augment the Datalog program with counters, which yields the following:

$axiom(A, B, c_1 \times c_2) : -gn(A, C, c_1), gv(C, B, c_2).$
$gn(A, B, c_1 \times c_2 \times c_3) : -art(A, C, c_1), adjs(C, D, c_2), n(D, B, c_3).$
$gn(A, B, c_1 \times c_2) : -art(A, C, c_1), n(C, B, c_2).$
$adjs(A, B, c_1 \times c_2) : -adjs(A, C, c_1), adj(C, B, c_2).$
$adjs(A, B, c_1) : -adj(A, B, c_1).$
$gv(A, B, c_1) : -v(A, B, c_1).$
$n(A, B, c_1) : -'D'(elephant, A, B, c_1).$
$art(A, B, c_1) : -'D'(the, A, B, c_1).$
$adj(A, B, c_1) : -'D'(little, A, B, c_1).$
$adj(A, B, c_1) : -'D'(green, A, B, c_1).$
$adj(A, B, c_1) : -'D'(greedy, A, B, c_1).$
$v(A, B, c_1) : -'D'(flies, A, B, c_1).$
$'D'(the, 0, 1, 1).$
$'D'(little, 1, 2, 1).$
$'D'(green, 2, 3, 1).$
$'D'(elephant, 3, 4, 1).$
$'D'(flies, 4, 5, 1).$

Executing the above Proof Counting Algorithm, we obtain in the set of theorems S^c the following relations (as a convention, we denote the relation that a predicate is interpreted as by capitalizing the first letter of the predicate name).

$S^c =$
$\{'D'(the,0,1,1),'D'(little,1,2,1),'D'(green,2,3,1),'D'(elephant,3,4,1),$
$'D'(flies,4,5,1)\} \cup \{V(4,5,1)\} \cup \{Adj(1,2,1), Adj(2,3,1)\} \cup \{Art(0,1,1)\} \cup$
$\{N(3,4,1)\} \cup \{Gv(4,5,1)\} \cup \{Adjs(1,2,1), Adjs(2,3,1), Adjs(1,3,1)\} \cup$
$\{Gn(0,4,1)\} \cup \{Axiom(0,5,1)\}.$

Since axiom(0,5) is present with the counter value 1, we conclude that the sentence in question is syntactically correct.

If we are now to analyze another similar sentence

'the little greedy elephant flies'

where only the word 'green' is changed to 'greedy'. We have

$$\Delta EDB^c = \{'D'(green,2,3,-1),'D'(greedy,2,3,1)\}.$$

In the application of the Counter Maintenance Algorithm, sentence 1) adds

$$\{'D'(green,2,3,-1),'D'(greedy,2,3,1)\}$$

to ΔS^c. The first iteration of the 'while' loop only produces

$$\{Adj(2,3,-1), Adj(2,3,1)\};$$

by rectification, we realize that ΔS^c will be empty at the end of this iteration. Thus, the computation terminates at once. And, the new set of theorems will be:

$S^c = S^c \uplus \{'D'(green,2,3,-1),'D'(greedy,2,3,1)\} =$
$\{'D'(the,0,1,1),'D'(little,1,2,1),'D'(greedy,2,3,1),'D'(elephant,3,4,1),$
$'D'(flies,4,5,1)\} \cup \{V(4,5,1)\} \cup \{Adj(1,2,1), Adj(2,3,1)\} \cup \{Art(0,1,1)\} \cup \{N(3,4,1)\} \cup$
$\{Gv(4,5,1)\} \cup \{Adjs(1,2,1), Adjs(2,3,1), Adjs(1,3,1)\} \cup$
$\{Gn(0,4,1)\} \cup \{Axiom(0,5,1)\}.$

Just as before, the second sentence is syntactically correct because axiom(0,5) is present with the counter value 1 in the set of theorems of the corresponding deductive database.

From the above example, one can recognize that in treating the sentences by Counter Maintenance Algorithm, only 'differential' efforts are performed on the basis of the results of analyzing other similar sentences.

These differential efforts may, of course, affect more of S^c than just words and their immediate categories. If, for instance, we first parse *the blue sky shines*, and then we incrementally update the semantics in order to parse *"the blue sounds sad"* (cf. Section 4), the Noun Phrase relation will be updated as well.

Notice also that ambiguous parses, while not directly, are accounted for in our approach, and that the degree of ambiguity (i.e., the number of parses for a given sentence) corresponds to the number of proofs, which is automatically obtained by the algorithm.

6 Performance results

The performances of DLGs are visibly better than those of DCGs under the OLDT-resolution based XSB-Prolog system [18] because atomic DLG-arguments have much smaller tabulation overhead than compound DCG-arguments.

For example, in the case of the grammar:

```
axiom-->s,
s --> [],
s --> s, [a], s, [a].
```

on a query containing 32 a symbols the execution time for 1000 iterations was 0.291s for DLGs compared with 0.530s for DCGs [8].

This difference in performance comes from a decrease in the theoretical complexity, as tabulated DCG arguments are in average proportional in size with the length of the input string while they are constant size in the case of DLGs. Let T be the number of tabulation operations and N the length of the input string. We have:

$DCG_{time} = O(T * N)$

and

$DLG_{time} = O(T * 1)$.

Note that if we add extra arguments which may build structures on top of a DLG grammar this advantage seems to become less important. However, the extra price will be paid only for the extra arguments and not for the state of the grammar arguments. Moreover, the size of extra arguments tend to be small compared with the size of the input sentence so that performance improvements can still be expected in this case.

The programmer can also decide, in a system like XSB-Prolog, which predicates will be tabulated and limit such operations to rules using left recursion, for instance.

7 Limitations of Datalog Grammars

Some of the DCG-based programming techniques that make assumptions about the list-based representation of `connect/3` are not *'portable'* to a DLG representation.

Let us consider the following DCG-based permutation program:

```
perm(Xs,Ys):-perm(Xs,[],Ys).

perm([])-->[].
perm([X|Xs])-->perm(Xs),ins(X).

ins(X),[X]-->[].
ins(X),[Y]-->[Y],ins(X).
```

Under standard DCG-expansion, this gives:

```
perm(A,B) :-  perm(A,[],B).

perm([],A,A).
perm([A|B],C,D) :- perm(B,C,E),ins(A,E,D).
```

[8] With XSB-Prolog 1.2 on a Sparcstation 10-40.

```
ins(A,B,C) :- 'C'(C,A,B).
ins(A,B,C) :- 'C'(B,D,E),ins(A,E,F),'C'(C,D,F).
```

Clearly, by looking at the ins/3 clause and by unfolding the `'C'/3` predicates one can see that in this case lists are essential.

The program consumes the list in its first argument and builds as its second DCG-argument a permuted list starting from [].

However, the problem does not come from the presence of the terminals in the head of the rule as one might think.

In a program like

```
rev(Xs,Ys):-rev(Xs,Ys,[]).

rev([])-->[].
rev([X|Xs])-->rev(Xs),[X].
```

terminals occur only in the body but again the program builds the reversed list using its first DCG-argument as an accumulator and making use of 'C'/3 essentially as a list constructor.

The same problem arises in the habitual implementations of more sophisticated types of logic grammars, such as Extraposition grammars [11], or Discontinuous grammars [1], which also typically rely on the use of `'C'/3` as a list constructor.

Sufficient conditions for equivalence of the DLG and DCG translations.

The following gives an (obvious) sufficient condition for the equivalence of the DLG-DCG translations.

Theorem 6 *If a program containing grammar rules verifies the following conditions:*

1. *it does not use more than one left-hand side symbol*
2. *no predicate definition originating from a grammar rule can also contain directly defined Horn clauses.*

then its DCG and DLG translations are operationally equivalent.

Proof. It follows from the fact that if the conditions are fullfiled, there's no way to make use of the concrete representation of the hidden arguments if the connect/3 predicate.

Remark that such as a grammar is still quite powerful as for instance extra arguments can still be present and used for various purposes.

8 Related Work

The closest system to the ideas explored in this paper is XSB [18], in which we can re-define the `'C'/3` predicate by representing a sentence as:

```
c(0,word1,n).
...
c(n-1,wordn,n).
```


XSB will issue a warning which can simply be ignored, and the result is in principle a chart parser. However, this approach has several drawbacks re. efficiency, as discussed in a March 30 1994 message from David Scott Warren to Richard O'Keefe on the Prolog net (comp.lang.prolog).

In the deductive database community, several authors have considered the problem of maintaining predicates defined through general rules. For example, [10], [19] studied the incremental evaluation of derived predicates as well by proof number memorization. However, the number they stored for each theorem is the number of 'one-step' proofs, hence some proof derivations can be avoided. But set-oriented maintenance is not supported explicitly there. [7] dealt with set-oriented update in three steps: one for the subset of insertions and two for that of deletions. Insertion treatment consists of re-executing the semi-naive evaluation algorithm by initializing 'Δ' to the subset of inserted facts. As for deletions, the set of deleted theorems is first over-computed by the semi-naive evaluation algorithm, with 'Δ' initialized to the subset of deleted facts; taking this over-computed set of theorems as a filter, the second step restores theorems deleted at the first step, but still derivable with the updated data.

9 Conclusion

We have presented a new type of logic grammar, Datalog grammars, which, in conjunction with appropriate compilation techniques, e.g. OLDT-resolution or Datalog, results in increased efficiency (with respect to that of traditionally compiled DCGs) while also ensuring termination.

We have also discussed its applications to natural language processing, such as parsing incomplete input. We have introduced an incremental algorithm for dealing with incomplete input in a fashion that avoids recomputation while examining the different alternatives.

We have also shown that DLGs can be extended to allowing function symbols in arguments with no loss of the termination property. This is important for many natural language applications, in which structure-building is desirable for such purposes as constructing semantic or syntactic representations for a sentence as a side-effect of parsing it.

It is also interesting to note that the techniques developed in this article make it possible to implement human language consultable deductive database systems which treat both components, i.e. the deductive database component and the natural language processing component, in a uniform as well as efficient manner.

Finally, we argue that the approach is interesting for big-scale natural language processing. First, the incremental technique proposed is optimal for large databases; second, constant-time OLD tabulation is ensured (in contrast with DCGs which would require linear effort in tabulating lists); last, polynomial execution time through Datalog evaluation techniques is ensured, in contrast to exponential time for conventional DCG evaluation in Prolog.

Acknowledgement

This research was supported by NSERC Operating grant 31-611024, and by NSERC, CSS and SFU PRG Infrastructure and Equipment grant given to the Logic and Functional Programming Laboratory at SFU, in whose facilities this work was developed. We are also grateful to the Centre for Systems Science, LCCR and the School of Computing Sciences at Simon Fraser University for the use of their facilities. Paul Tarau thanks for support from the Canadian National Science and Engineering Research Council (grant OGP0107411) and the FESR of the Université de Moncton.