Proc. 3rd Int. Conf. on Data and Knowledge Bases, (C.Berri, J.W.Schmidt, U.Dayal eds.), Jerusalem (Israel), 28-30 June 1988, Morgan Kaufmann, pp.389-404.

Datalog Automata

Extended Abstract

Bernard LANG
INRIA
B.P. 105, 78153 Le Chesnay, France
lang@inria.inria.fr

Abstract

We propose a new computational paradigm for the evaluation of recursive Datalog queries, which is based on a pushdown automaton (PDA) model. By extending to these automata a dynamic programming technique developed for PDAs in context-free parsing, we obtain a general and simple technique for constructing efficient polynomial query evaluators.

Keywords: Datalog, Recursive Queries, Complete Strategies, Dynamic Programming, Polynomial Complexity.

1 Introduction

A large number of strategies have been developed for the optimization of Datalog queries. The diversity and complexity of these strategies makes it rather difficult to organize them into a unified framework, permitting easy comparison and performance analysis [2]. Our aim here is an attempt at some unification based on the parenthood of Datalog query processing and *context-free (CF)* parsing, which has already been noted, and drawn upon, by several authors [25, 27][1]. We shall be mostly concerned with strategies that are *complete* in Datalog, i.e. that always terminate and produce *all* answers to a given query (and not just one answer).

When designing, proving or analyzing Datalog programs, one has to worry simultaneously about the soundness and completeness of the evaluation strategy, and about the structures needed to implement that strategy efficiently — which usually means parsimonious use of backtracking — so as to handle non-determinism and to store variable bindings. We propose here to separate the two concerns by proving soundness and completeness on an intermediate non-deterministic device, the *DataLog Automaton (DLA)*, which embodies the essence of the query evaluation strategy. We then present a standard dynamic programming technique to handle the non-determinism and the representation of variable bindings, which gives an effective and efficient implementation of DLAs.

This work is an extension of previous results of the author [16] in which a similar approach is used to build variants of Earley's algorithm [7] for context-free parsing. The raw algorithm as designed by Earley is notoriously too inefficient for practical applications. On the other hand, a variety of very efficient parsing algorithms have been designed by the compiler community (e.g. [1, 5]), but they are usually considered applicable only to restricted subclasses of the context-free languages on which they all follow a deterministic *Push-Down Automaton (PDA)* model.

As it turns out, all classical parser construction techniques are still correct outside their recognized context-free class, but they lose the determinism of the constructed PDA. The contribution of [16] was to define an Earley-like construction for non-deterministic PDAs. It thus provides the means for turning those non-deterministic but efficient (when making the right choices) PDAs into efficient all-paths parsing algorithms. This approach gives a unified view of many attempts at specific optimizations of, or variations on Earley's algorithm [4, 28, 32, 26, 33], most of which may be seen as application of Earley's dynamic programming construction to a specific kind of PDA[2]. The PDA construction embodies the parsing strategy [9, 13], while the Earley-like construction takes care of non-determinism.

Our generalization of this work to recursive queries is very similar in its essence to the work of Vieille [37, 38], both with respect to its dynamic programming foundation and with respect to its stated objective of obtaining a unified view of recursive query processing.

Earley's algorithm is itself a specific implementation of a more general dynamic programming parsing tech-

[1] The mini-language Proplog used in [22] as an introduction to Datalog and Prolog is very similar to a CF language in which the input symbols can be scanned in random order.

[2] These algorithms are sometimes incomplete with respect to infinite ambiguity of the input sentence, i.e. they do not quite correspond to the complete Earley construction [28, 32].

nique known as CYK (for Cocke, Younger and Kasami [10, 40, 12]). Essentially, Earley's construction is a variant of CYK in which the order for computing intermediate results (i.e. sub-parses, or instantiated relations) has been fixed [28, 30, 8], and some predictive optimization has been added. Even then, the CYK algorithm may be itself seen as constructed from a PDA, usually trivially built from the language grammar, and following a bottom-up strategy. The elements memorized by the dynamic programming procedure are essentially the items [1] — or states [7] — of Earley's algorithm.

Some query evaluation techniques (e.g. *active connection graphs* [21] or *system graphs* [14]) fall into the more general CYK paradigm since they do not propose an evaluation order, generally because they are intended for distributed (or multi-) processing, similar to the multiprocessor Earley parser of [19]. However the query optimizations used in this context — as well as in others — seem transposable to the PDA approach, and we believe it to be an adequate computational paradigm on sequential architectures.

The Datalog Automaton, on which this paper is based, is essentially a PDA that stores facts rather than simple symbols (i.e. argument-less predicates) on its push-down store. This is not surprising: the nondeterministic PDA is the natural device to explore an AND-OR tree (as one may view the computation of a query [34]), with the stack to take care of AND nodes and the non-determinism to handle OR nodes.

In section 2, we give the basic definitions for Datalog programs and automata. In sections 3 we show that any Datalog automaton computes the answers to a recursive query, and that conversely recursive queries may be answered by Datalog automata. Section 4 describes the transformation of a Datalog automaton into a deterministic algorithm, and it is followed by a simple complexity analysis in section 5. In the appendix, the relation between CF parsing and recursive queries is developed in more details.

The reader is supposed to be familiar with the basics of context-free languages, push-down automata, relational databases and recursive queries.

2 Definitions

2.1 Datalog Programs

To emphasize the parenthood between Datalog and context-free languages, we shall use indifferently the terminology of both fields. This is justified in more details in the appendix. A Datalog *program* or *DataLog Grammar (DLG)* corresponds to a context-free *grammar*, *extensional relations* (or *predicates*[3]) correspond to CF *terminals*, *intensional relations* correspond to CF *nonterminals*, Datalog *clauses* are the equivalent of CF *production* or *reduction rules*. The *query* is the CF grammar *axiom*. We assume, without any loss of generality (even with respect to complexity or practical efficiency) that the query is reduced to a single literal containing only uninstantiated variables.

We shall consider several sets of predicate symbols. Each predicate symbol has a non-negative integer *arity* (number of arguments) given by a function `arity` defined on all the predicates to be introduced.

A Datalog grammar $\mathbf{G}$ is a 6-tuple:

$$\mathbf{G} = (\mathbf{X}, \mathbf{C}, \mathbf{\Sigma}, \mathbf{V}, \mathbf{\Pi}, \overset{\circ}{\mathrm{N}}\overset{\circ}{x})$$

where:

- $\mathbf{X}$ is a denumerable and *ordered* set of *variables*.
- $\mathbf{C}$ is a finite set of *constant* symbols.
- $\mathbf{\Sigma}$ is a finite set of terminal relations symbols, i.e. extensional predicates.
- $\mathbf{V}$ is a finite set of nonterminal relations symbols, i.e. intensional predicates.
- $\mathbf{\Pi}$ is a finite set of $\mathbf{n}$ *clauses* of the form $\mathrm{N}t$ `:-` η where the *head* $\mathrm{N}t$ is an intensional literal, and the *body* η is a finite sequence of literals.

  There may be several clauses with the same head.
- $\overset{\circ}{\mathrm{N}}\overset{\circ}{x}$ is the *initial literal*, i.e. the *query*.

A *literal* is a predicate symbol A applied to variable or constant arguments in number equal to `arity`(A). The literal is a said to be a *syntactic fact* or a *ground literal*. if it has only constant arguments. The arguments will be usually lumped into a vector of arguments. Such vectors will be called *tuples*. We shall use the characters x y and z for tuples of variables, t u and v for tuples mixing variables and constants, and a b and c for tuples of values of the interpretation domain (see below). We use *subscripting*, as in u_i and u_j, to denote distinct tuples, and not for selection of a tuple component (i.e. indexing). When the arguments of its predicate are represented by a tuple, a literal is noted by juxtaposition as in: $\mathrm{A}t$, for predicate A and argument tuple t.

We shall not distinguish, in notation or otherwise, a set of variables and the tuple of the same variables in the canonical order of $\mathbf{X}$ and without repetition. Thus set operations may be applied to such tuples of variables. We note $\mathbf{X}(\heartsuit)$ the set or tuple of variables occurring in $\heartsuit$, where the symbol $\heartsuit$ stands for any tuple, literal, clause, etc., or sets and sequences thereof. The *empty tuple* is noted ϵ. It is usually omitted in literals with nullary predicates.

[3] We shall use the words *relation* and *predicate* as synonyms.

An *interpretation* for the above defined DLG $\mathbf{G}$ is a pair:

$$\mathcal{I} = (\mathcal{D}_\mathcal{I}, \phi_\mathcal{I})$$

where:

$\mathcal{D}_\mathcal{I}$ is a finite set of values, called the *interpretation domain*.

$\phi_\mathcal{I}$ is a function that maps $\mathbf{C}$ into $\mathcal{D}_\mathcal{I}$, and maps $\mathbf{\Sigma}$ into the relations on $\mathcal{D}_\mathcal{I}$ with preservation of the arity, i.e.[4]

$$\forall \mathrm{A} \in \mathbf{\Sigma},\ \ \phi_\mathcal{I}(\mathrm{A}) \in \mathcal{P}(\mathcal{D}_\mathcal{I}^{\mathtt{arity}(\mathrm{A})})$$

The function $\phi_\mathcal{I}$ is called the *interpretation function*.

A *valuation*[5], or *variable assignment*, or *variable binding*, of the variables in $\mathbf{X}$ on $\mathcal{D}_\mathcal{I}$ is a mapping ν from $\mathbf{X}$ into $\mathcal{D}_\mathcal{I}$. Since we usually consider only finite subsets of $\mathbf{X}$, we define valuations only by their restriction to the subset considered. Given the interpretation $\mathcal{I}$, a valuation on $\mathcal{D}_\mathcal{I}$ has a canonical extension on $\mathbf{X} \cup \mathbf{C}$ by taking its union with the function $\phi_\mathcal{I}$. It can be further extended in the standard way to tuples in $\mathbf{X} \cup \mathbf{C}$, and then as a function from literals to facts.

An interpretation may be extended into a *minimal model* that maps also the functions of $\mathbf{V}$ into the relations on $\mathcal{D}_\mathcal{I}$, so that for any clause γ of the DLG $\mathbf{G}$ and for any valuation in $\mathcal{D}_\mathcal{I}$ of the variables in $\mathbf{X}(\gamma)$, if the interpretation of the tuples in the body belong to the relation interpreting the corresponding predicate symbol, then this is also true for the head tuple [35].

We call the interpretation the *actual data base*, and the tuples on $\mathcal{D}_\mathcal{I}$ are *actual tuples*. An interpreted literal (i.e. a predicate symbol applied to an actual tuple) is called a *fact*. A fact $\mathrm{A}a$ is said to be *true* iff $a \in \phi_\mathcal{I}(\mathrm{A})$.

Given a DLG $\mathbf{G}$ and an interpretation $\mathcal{I}$ for $\mathbf{G}$, we call the *answer* of $\mathbf{G}$ on $\mathcal{I}$ the set of tuples on $\mathcal{D}_\mathcal{I}$ in the relation $\phi_\mathcal{I}(\overset{\circ}{\mathrm{N}})$. This set is denoted by $\mathbf{G}(\mathcal{I})$. It may be computed with a variety of operational models, most of which are based on resolution [29, 20, 22].

Given an interpretation $\mathcal{I}$, a literal $\mathrm{A}t$ *matches* a fact $\mathrm{A}a$ in this interpretation iff there is a valuation ν such that $\nu(t) = a$. The fact $\mathrm{A}a$ is said to be an *instance* of the literal $\mathrm{A}t$, or more precisely a *fact instance*.

Given a literal $\mathrm{A}t$ and a valuation ν, we note $\nu(\mathrm{A}t)$ the application of A to the vector of values $\nu(t)$, i.e. $\mathrm{A}\nu(t)$ rather than the truth value of " $\nu(t) \in \phi_\mathcal{I}(\mathrm{A})$ " as is usual. The reason is that we need a convenient notation to talk about facts[6].

[4] $\mathcal{P}(S)$ denotes the set of subsets of S.

[5] A valuation is similar to an instantiation that substitutes a constant to every variable to produce a ground literal, but it uses interpretation values.

[6] This notational difficulty is due to the fact that the automaton actually manipulates values in the interpretation domain, rather than syntactic objects as is usual and preferable. We chose to do that so as to make clear that the compilation of intensional relations and queries can be done independently of a changing database, without introducing too many domains and mappings. This issue is also discussed in [22, page 181] with a slightly different terminology.

2.2 Datalog Automata

A Datalog Automaton is essentially a pushdown automaton that stores facts in its pushdown store, instead of argument-less predicate symbols[7]. This may be viewed as a structured stack alphabet, in the same way that Datalog grammars may be seen as CF grammars with structured sets of terminals and nonterminals[8].

A secondary point is that we use *stateless* automata. This is achieved without loss of generality by defining popping transitions as replacing the top two facts of the stack by only one fact. Though states are very convenient to simplify some theoretical constructions, they are often not really needed in practice. This is typically the case for the most powerful class of deterministic CF pushdown parsers: the LR(k) family [5].

A DLA $\mathcal{A}$ is defined as a 7-tuple:

$$\mathcal{A} = (\mathbf{X}, \mathbf{C}, \mathbf{\Sigma}, \mathbf{\Delta}, \overset{\circ}{\$}, \$_\mathrm{f}, \delta)$$

where:

$\mathbf{X}$ is a denumerable and *ordered* set of variables.

$\mathbf{C}$ is a finite set of constant symbols.

$\mathbf{\Sigma}$ is a finite set of terminal relations symbols, i.e. extensional predicates.

$\mathbf{\Delta}$ is a finite set of relation symbols used for the facts stored in the stack.

$\overset{\circ}{\$}$ is the *initial fact*, i.e. the only fact initially in the stack. The nullary predicate $\overset{\circ}{\$}$ is called the *initial predicate*.

$\$_\mathrm{f}$ is the *final predicate*.

δ is a finite set of *transitions* described below.

The transitions in δ come in three kinds:

horizontal transitions:	$(\mathrm{F}t\ \mathrm{A}u \mapsto \mathrm{G}v\ )$
push transitions:	$(\mathrm{F}t\ \mathrm{A}u \mapsto \mathrm{G}v\ \mathrm{F}t)$
pop transitions:	$(\mathrm{F}t\ \mathrm{G}u \mapsto \mathrm{H}v\ )$

[7] An extension of the formalism presented here is being developed [18], which allows variables in stack literals. This increases the power of the formalism (e.g. for handling infinite interpretation domains) and allows some optimizations. However it requires replacing equality with matching, which complicates the constructions.

[8] This set is even infinite in the case of Prolog (see also for example [31]).

where:
$\mathrm{F}, \mathrm{G}, \mathrm{H} \in \mathbf{\Delta}$, $\mathrm{A} \in \mathbf{\Sigma} \cup \{\diamond\}$, and $t, u, v \in (\mathbf{X} \cup \mathbf{C})^\star$. The symbol $\diamond$ is a special nullary predicate indicating that the transition does not depend on any extensional relation[9].

Interpretations are defined for DLAs exactly in the same way as for Datalog grammars. We now define the *computation* of the DLA $\mathcal{A}$ for an interpretation $\mathcal{I} = (\mathcal{D}_\mathcal{I}, \phi_\mathcal{I})$.

A configuration of the DLA $\mathcal{A}$ is just a stack of facts built with the stack predicates of $\mathbf{\Delta}$ and tuples from the interpretation domain $\mathcal{D}_\mathcal{I}$. In the sequel, the stack will usually be represented as a sequence of literals, *with the top of the stack on the left-hand side.*

A horizontal or push transition, as denoted above, is *applicable* to a stack ξ iff there is a valuation ν in $\mathcal{D}_\mathcal{I}$ through which $\mathrm{F}t$ matches the top literal of the stack, and $\mathrm{A}u$ matches a fact in $\mathcal{I}$. A pop transition is applicable if there is a valuation ν such that the literals $\mathrm{F}t$ and $\mathrm{G}u$ match respectively the first and second literals of the stack[10].

When the extensional literal of a transition is $\diamond$, it may simply be ignored in the above definitions.

Applying an applicable transition consists in popping from the stack the one or two facts that have been matched, and replacing them by the new facts obtained by application of the valuation ν to the literals on the right-hand side of the transition. Note that in the case of the push transition, the former top literal $\nu(\mathrm{F}t)$ is not really removed but is only covered with the new top $\nu(\mathrm{G}v)$.

A *final configuration* is a stack composed of only one fact which is a terminal fact, i.e. with the predicate $\$_\mathrm{f}$. A computation *terminates successfully* when it reaches a final configuration. The $\mathcal{D}_\mathcal{I}$-tuple of the unique stack fact is the *answer of that computation.* A computation may also fail to terminate successfully, either because it applies transitions without ever reaching a final configuration, or because it remains blocked in a non-final configuration without any applicable transition. These two cases are respectively called *non-terminating* and *failing* computations.

The computation of a DLA is *non-deterministic* in 2 ways:

1. in the choice of the applied transition, when several are applicable;
2. in the choice of the valuation ν. This choice is limited by the necessity to match the stack fact(s), but the remaining freedom may be seen again as two choices:
   - choice of the fact from the interpretation matched for horizontal and push transition,
   - arbitrary choice of the valuation of variables on the right-hand side of the transition that do not appear on the left-hand side (if any).

The *answer* of a DLA $\mathcal{A}$ on the interpretation $\mathcal{I}$ is the set of all answers that may be produced by some computation of $\mathcal{A}$ on $\mathcal{I}$. This set is noted $\mathcal{A}(\mathcal{I})$.

[9] It may also be viewed as a nullary relation that contains the empty tuple ϵ in all interpretations.
A transition that uses (resp. does not use) an extensional relation is said to be *scanning* (resp. *non-scanning*).

[10] The absence of extensional predicate for pop transitions is only for expository convenience and implies no loss of generality.

3 Equivalence of DLG and DLA

The Datalog grammar is essentially a denotational or generative formalism, while the Datalog automaton is rather an operational formalism, intended to model a query processing device. In this section we show that these two formalisms have the same power, i.e. that for each construct of one family, there is an object of the other family that gives the same answers on the same interpretations.

3.1 Reduction of DLA to DLG

From a practical point of view, the reduction of DLA to DLG is not essential since we are more interested in building computational devices from declarative definition than in the converse. It is mentioned here for the sake of completeness of the exposition.

Proposition 3.1 *For every DLA $\mathcal{A}$ there is a DLG $\mathbf{G}$ with the same constants and extensional predicates, such that for any interpretation $\mathcal{I}$ they both give the same answer, i.e.* $\mathbf{G}(\mathcal{I}) = \mathcal{A}(\mathcal{I})$

3.2 Reduction of DLG to DLA

We shall give here a rather naive construction of a DLA from a given DLG. This construction is intended only as an existence proof, to justify the determinization algorithm given in the next section. Of course, we expect that the construction of practical query compilers following our technique will use sophisticated optimizations when constructing the DLA, similar to those already existing in the literature [34, 14, 2]. Our purpose in this paper is only to show how to separate the construction (and optimization) of a DLA from its transformation into a deterministic algorithm by a standardized (and optimized) technique.

Nevertheless, we shall see that this naive construction is already sufficient to give good complexity results in the final deterministic algorithm.

Proposition 3.2 *For every DLG* $\mathbf{G}$ *there is a DLA* $\mathcal{A}$ *with the same constants and extensional predicates, such that for any interpretation* $\mathcal{I}$ *they both give the same answer, i.e.* $\mathbf{G}(\mathcal{I}) = \mathcal{A}(\mathcal{I})$

We consider the DLG $\mathbf{G} = (\mathbf{X}, \mathbf{C}, \mathbf{\Sigma}, \mathbf{V}, \mathbf{\Pi}, \overset{\circ}{\text{N}}x)$. From $\mathbf{G}$ we build a DLA $\mathcal{A}$ that uses a naive non-deterministic bottom-up strategy to prove the goal.

Let $\mathbf{\Pi}$ be a set of $\mathbf{n}$ clauses $\{\gamma_k \mid 1 \leq k \leq \mathbf{n}\}$ where the clause γ_k has the form:

$$\gamma_k : \quad \text{F}_{k,0}\, t_{k,0} \text{ :- } \text{F}_{k,1}\, t_{k,1}, \ldots, \text{F}_{k,i}\, t_{k,i}, \ldots, \text{F}_{k,n_k}\, t_{k,n_k}$$

In our construction we assume that the literals in the body are in a fixed order from left to right, as indicated by the index i.

The set $\mathbf{\Delta}$ of stack predicates is defined as:

$$\mathbf{\Delta} \;=\; \{\overset{\circ}{\$}\} \cup \hat{\mathbf{\Sigma}} \cup \hat{\mathbf{V}} \cup \text{Pos}$$

where:

$\overset{\circ}{\$}$ is a new nullary predicate symbol.

$\hat{\mathbf{\Sigma}}$ is a set of new predicate symbols similar to $\mathbf{\Sigma}$: $\hat{\mathbf{\Sigma}} = \{\hat{\text{A}} \mid \text{A} \in \mathbf{\Sigma}\}$.

$\hat{\mathbf{V}}$ is a set of new predicate symbols similar to the set $\mathbf{V}$[11]: $\hat{\mathbf{V}} = \{\hat{\text{A}} \mid \text{A} \in \mathbf{V}\}$.

Pos is a set of new predicate symbols characterized by 2 integers k and i, noted $\bigtriangledown_{k,i}$, and denoting positions between literals of the clauses[12]:

$$\text{Pos} \;=\; \{\bigtriangledown_{k,i} \mid 1 \leq k \leq \mathbf{n},\ 0 \leq i \leq n_k\}$$

The arity of these symbols is chosen to fit the size of their argument vectors as specified below.

We also define the final predicate as $\$_\text{f} = \overset{\circ}{\text{N}}$. Then we build the DLA

$$\mathcal{A} = (\mathbf{X}, \mathbf{C}, \mathbf{\Sigma}, \mathbf{\Delta}, \overset{\circ}{\$}, \$_\text{f}, \delta)$$

where the set δ contains the transitions defined below in the informal description of the operating strategy of the automaton.

The "bottom-up" DLA works in two modes: *scanning* and *reduction*.

[11]The set $\hat{\mathbf{V}}$ is not really necessary and is introduced only for uniformity of notations; $\mathbf{V}$ could be used directly. More generally, "*hatting*" is used for purely formal reasons, viz. keeping separate the sets of extensional and stack predicates. It need not be reflected in an implementation.

[12]These predicates are very similar to the *dotted rules* used in Earley's states [7], or to the states of Knuth's original LR(k) parser [15].

In the scanning mode, the DLA randomly pushes facts from the database onto its stack, assuming that (with luck) they will be needed in reverse order for the proof. This is achieved by the following push transitions:

$$\forall \text{M} \in (\overset{\circ}{\$} \cup \hat{\mathbf{\Sigma}} \cup \hat{\mathbf{V}}),\ \forall \text{A} \in \mathbf{\Sigma} : \qquad (\text{M}x\ \text{A}y \;\mapsto\; \hat{\text{A}}y\ \text{M}x) \tag{1}$$

The predicate $\hat{\text{A}}$ is similar to A, but it is hatted to indicate that it is now a stack predicate, rather than an extensional predicate. The vectors x and y are arbitrary vectors of all different variables. with adequate size. The DLA computation starts with one of the above transitions, where $\text{M} = \overset{\circ}{\$}$.

At any time in scanning mode, the DLA may decide to prove some instance of the head $\text{F}_{k,0}\, t_{k,0}$ of a clause γ_k, using the top n_k facts of the stack. This is initialized by pushing on top of the stack the *nullary* predicate $\bigtriangledown_{k,n_k}$ by means of one of the following push transition:

$$\forall \text{M} \in (\overset{\circ}{\$} \cup \hat{\mathbf{\Sigma}} \cup \hat{\mathbf{V}}),\ \forall k \in [1, \mathbf{n}] : \qquad (\text{M}x \diamond \;\mapsto\; \bigtriangledown_{k,n_k}\, \text{M}x) \tag{2}$$

When the stack top has a predicate in Pos, the DLA is in reduction mode. If the stack top predicate is $\bigtriangledown_{k,i}$, this means that the subgoals $\text{F}_{k,i+1}\, t_{k,i+1}$ to $\text{F}_{k,n_k}\, t_{k,n_k}$ of the clause γ_k have already been proved. The argument vector of the stack top fact is precisely (at least) the variable bindings (valuation) resulting from these proofs, which will have to be used when proving the remaining subgoals, and then when finally valuating the head of the clause.

The proof of the next rightmost literal $\text{F}_{k,i}\, t_{k,i}$ is then attempted by trying to match it with the second (from top) fact on the stack, using one of the following pop transitions:

$$\forall k \in [1, \mathbf{n}],\ \forall i \in [1, n_k] : \qquad (\bigtriangledown_{k,i} x_{k,i}\ \hat{\text{F}}_{k,i}\, t_{k,i} \;\mapsto\; \bigtriangledown_{k,i-1}\, x_{k,i-1}\,) \tag{3}$$

The vectors of variables $x_{k,i}$ could be taken very simply to be the vector $\mathbf{X}(\gamma_k)$ of all variables occurring in the clause γ_k. However, to obtain a somewhat better complexity result for our final algorithm, we choose a tighter definition[13]. Thus we restrict this vector to those variables of γ_k that appear in the already proved part of the body (i.e. in $t_{k,i}$ to t_{k,n_k}), and whose established bindings will be needed when proving the rest of the body and instantiating the head:

$$x_{k,i} \;=\; \bigcup_{j=i+1}^{n_k} \mathbf{X}(t_{k,j}) \;\cap\; \bigcup_{j=0}^{i} \mathbf{X}(t_{k,j})$$

[13]This is the only place where we try to be a little more clever.

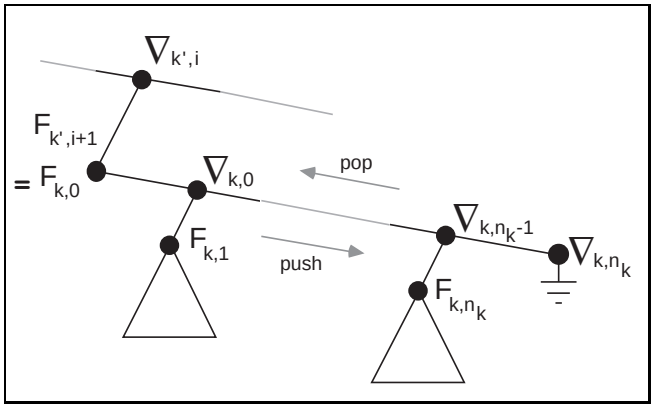


Figure 1: Representation of a fragment of the proof tree corresponding to the DLA computation. Note that the body of a clause (i.e. an AND node) is represented by a Lisp-like list of its literals, ending with an equivalent of the Lisp NIL.

When the stack top predicate is $\bigtriangledown_{k,0}$, this means that some instance of the body of the clause γ_k has been proved. The top literal of the stack is then replaced by the instantiated head of γ_k using one of the following horizontal transitions:

$$\forall k \in [1, \mathbf{n}] : \quad (\bigtriangledown_{k,0} x_{k,0} \diamond \mapsto \mathrm{F}_{k,0} t_{k,0}) \qquad (4)$$

Then the DLA is back in scanning mode.

Note that if a variable occurs in the head $\mathrm{F}_{k,0} t_{k,0}$ of the clause γ_k, but not in its body, then it does not occurs in $x_{k,0}$ as defined above. When applying the transition, the automaton chooses non-deterministically an arbitrary binding for this variable. This corresponds precisely to the arbitrary binding of a head variable that does not occur in the body of a clause[14].

The computation ends successfully when the stack contains only one fact that is an instance of the query $\mathring{\mathrm{N}}\mathring{x}$.

We do not give here the proof that for any interpretation $\mathcal{I}$ we have $\mathbf{G}(\mathcal{I}) = \mathcal{A}(\mathcal{I})$. Such a proof relies on the fact that a successful computation of the automaton is isomorphic to a postfix walk on (an encoding of) a proof tree, as partially represented in figure 1.

4 Determinization of a DLA Computation

Having established that a DLG **G** may always be transformed into a DLA $\mathcal{A}$, we now proceed to show how a DLA may be simulated by a *complete* deterministic algorithm that computes the answers of all computations by simulating all these computations simultaneously[15]. An important feature of this algorithm is that it always terminates, independently of the original DLG or DLA. Furthermore, by appropriate sharing of common subcomputations, it achieves quite good theoretical complexity bounds as will be described in the next section.

This algorithm is an adaptation to DLAs of an algorithm developed by the author in [16] for nondeterministic PDAs, which is itself an adaptation to PDAs of Earley's grammar based technique for parsing CF languages [7].

The formal algorithm presented below is to be completed with a few implementation guidelines given with the complexity analysis in section 5. Working out some of the details is essential, since we propose this construction as a standard component of query compilers, unlike the DLA construction which was only an example.

4.1 The basic data structure

The problem is to compute $\mathcal{A}(\mathcal{I})$ where $\mathcal{A}$ is a DLA $\mathcal{A} = (\mathbf{X}, \mathbf{C}, \mathbf{\Sigma}, \mathbf{\Delta}, \mathring{\$}, \$_{\mathrm{f}}, \delta)$ as defined in section 2[16]. This is done by constructing a collection of *items*[17] that play the same role as those of Earley, as specified in the algorithm below.

An item is a pair of facts representing a fragment of a computation of the DLA $\mathcal{A}$.

More precisely, an item $U = <\mathrm{F}a\ \mathrm{G}b>$ may be regarded as the equivalence class of all configurations (i.e. stack contents) of the DLA such that:

- The top of the stack is a fact $\mathrm{F}a$.
- The second stack fact is $\mathrm{G}b$.
- There is a DLA computation starting from a configuration with $\mathrm{G}b$ on top, which immediately pushes a new fact on it, and then reaches the configuration with $\mathrm{G}b$ and $\mathrm{F}a$ on top without ever uncovering $\mathrm{G}b$.

The proof that our dynamic programming interpretation of the DLA is sound and complete relies on two lemmas that formalize the above intuitive interpretation of items. Soundness and completeness follow.

In figure 2 we represent pictorially a collection of items. The facts are denoted by subscripted K's. We have added arrows from the second component of every item $<K_i\ K_j>$ to the first component of any item

[14] This introduces a considerable non-determinism which later-on will result in increased computation or even complexity. It could be avoided by allowing variables in stack tuples.

[15] As noted by Sheil in [30], the essential point in such a simulation is a proper choice of the data-structure used, and not the order in which it is created.

[16] This construction is independent of the way the DLA was obtained, though the DLA construction will be considered too for a finer complexity analysis.

[17] The word *item* is a terminology from [1] and others; Earley uses the word *state* in [7].

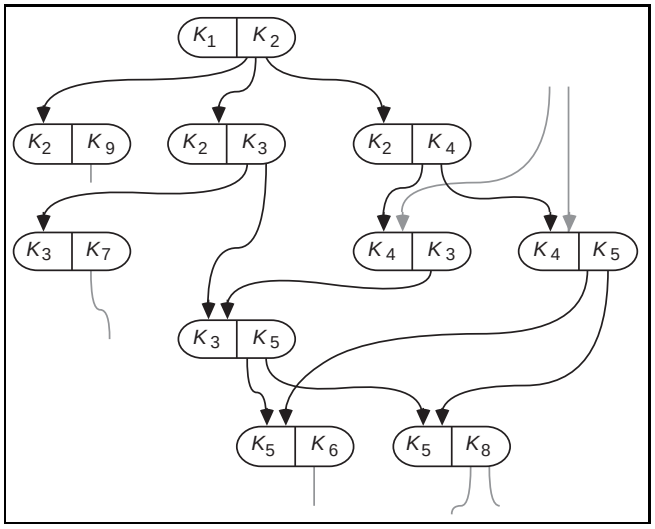


Figure 2: A collection of items represents possible stack configurations.

$<K_j\ K_k>$. This is to indicate that a possible stack configuration of a computation may be composed of the sequence of first facts of any sequence of thus chained items. Hence, if the collection of items of figure 2 is produced by a dynamic programming computation, it means that a standard non-deterministic computation of the DLA could have produced any of the following stack configurations (among others): $K_1K_2K_9\ldots$, $K_1K_2K_3K_7\ldots$, $K_1K_2K_3K_5K_6\ldots$, $K_1K_2K_3K_5K_8\ldots$, $K_1K_2K_4K_3K_5K_8\ldots$, $K_1K_2K_4K_5K_8\ldots$, and so on.

4.2 The algorithm

4.2.1 Initialization

The computation of the deterministic algorithm is initialized by creating the *initial item* $\overset{\circ}{U} = <\overset{\circ}{\$}\ \dashv>$, where $\dashv$ indicates the bottom of the stack, but may also be read as a special nullary relation.

Then new items are constructed by applying the transitions of the DLA to the already existing items as described below, until no new item is created. Since the number of distinct items is finite, *the algorithm always terminates*, (provided — as we shall show — that all steps are finite).

To each item $U = <\mathrm{F}a\ \mathrm{G}b>$ we apply the transitions of the DLA $\mathcal{A}$ as follows:

4.2.2 Horizontal transition:

$$(\mathrm{F}t\ \mathrm{A}u\ \mapsto\ \mathrm{H}v\ )$$

This transition is applicable to the item $U = <\mathrm{F}a\ \mathrm{G}b>$ iff there is a valuation ν such that $\mathrm{F}a = \nu(\mathrm{F}t)$ and $\mathrm{A}c = \nu(\mathrm{A}u)$, for some tuple c in the relation $\phi_{\mathcal{I}}(\mathrm{A})$.

When $\mathrm{A} = \diamond$ the second condition "$\mathrm{A}c = \nu(\mathrm{A}u)$" may be ignored.

Then, for all such valuations, we create a new item (unless it already exists):

$$V = <\nu(\mathrm{H}v)\ \mathrm{G}b>$$

Note that, since valuations range over the whole of $\mathbf{X}$, there are infinitely many valuations satisfying the above constraints. However, we are only interested in those that differ over the variables used in the transition, i.e. $\mathbf{X}(t, u, v)$. Since $\mathcal{D}_{\mathcal{I}}$ is finite, there are only finitely many valuations that differ on these variables. In practice these valuations are obtained by pattern matching between $\mathrm{F}t$ and $\mathrm{F}a$, and between $\mathrm{A}u$ et $\mathrm{A}c$.

These remarks on the valuations are also valid for the other kinds of transitions, and will not be repeated.

4.2.3 Push transition:

$$(\mathrm{F}t\ \mathrm{A}u\ \mapsto\ \mathrm{H}v\ \mathrm{F}t)$$

This transition is applicable to the item $U = <\mathrm{F}a\ \mathrm{G}b>$ iff there is a valuation ν such that $\mathrm{F}a = \nu(\mathrm{F}t)$ and $\mathrm{A}c = \nu(\mathrm{A}u)$ (when $\mathrm{A} \neq \diamond$), for some tuple c in the relation $\phi_{\mathcal{I}}(\mathrm{A})$.

Then, for all such valuations, we create a new item (unless it already exists):

$$V = <\nu(\mathrm{H}v)\ \mathrm{F}a>$$

4.2.4 Pop transition:

$$(\mathrm{F}t\ \mathrm{G}u\ \mapsto\ \mathrm{H}v\ )$$

This transition is applicable to the item $U = <\mathrm{F}a\ \mathrm{G}b>$ iff there is a valuation ν such that $\mathrm{F}a = \nu(\mathrm{F}t)$ and $\mathrm{G}b = \nu(\mathrm{G}u)$.

Then, for all such valuations, and *for all items* $W = <\mathrm{G}b\ \mathrm{M}c>$ we create a new item (unless it already exists):

$$V = <\nu(\mathrm{H}v)\ \mathrm{M}c>$$

It is essential to note that the above "*for all items*" means: *for all items already existing* **or to be created**.

4.2.5 Termination

The algorithm terminates when all applicable transitions have been applied to all produced items, and that all produced items that could be used in the "for all items" steps of pop transitions have been thus used.

Then the answer is the set of facts:

$$\mathcal{A}(\mathcal{I}) = \{\$_{\mathrm{f}}a \mid <\$_{\mathrm{f}}a\ \dashv> \textit{ is a computed item}\}$$

5 Complexity Analysis

5.1 Space complexity of the algorithm.

The space complexity of the algorithm is proportional to the number of items actually constructed by the algorithm (this definition is compatible with the implementation suggested below).

In the general case, i.e. for an arbitrary DLA, an obvious upperbound for the space complexity is

$$O(|\mathcal{D}_\mathcal{I}|^{2\mathbf{d}})$$

where $|\mathcal{D}_\mathcal{I}|$ is the cardinality of the interpretation domain $\mathcal{D}_\mathcal{I}$[18], and $\mathbf{d}$ is the maximum arity of a stack predicate. This results trivially from the fact that an item is composed of two stack facts, and the number of distinct stack facts is at most $|\mathbf{\Delta}| \times |\mathcal{D}_\mathcal{I}|^{\mathbf{d}}$.

However we can give a better bound with the bottom-up DLA of section 3.2. For this, we shall express the complexity in terms of the original Datalog grammar $\mathbf{G}$, which is the really interesting result when compiling queries. The construction of the DLA is such that the predicates $\bigtriangledown_{k,i}$ in Pos can only appear at the top of the stack, while the original predicates in $\mathbf{\Sigma} \cup \mathbf{V}$ can appear anywhere[19]. Thus the $\bigtriangledown_{k,i}$ predicates can only appear in first components of items, never as second components. Hence we have a space complexity

$$O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}+\mathbf{d}}) \tag{5}$$

where $\mathbf{a}$ is the maximum arity of the predicates (intensional or extensional) of the original Datalog grammar.

It may be noticed that this complexity does not depend on the maximum number of variables in a clause, but only, at worst, on the maximum arity of the $\bigtriangledown_{k,i}$ predicates in Pos, i.e. the maximum number $\mathbf{d}'$ of variable bindings that must be carried between the literals of a clause[20]. This is because $\mathbf{d} = max(\mathbf{a}, \mathbf{d}')$.

Note that we may have $\mathbf{d}' \geq \mathbf{a}$, but we always have $\mathbf{d}' \leq \mathbf{r}$, where $\mathbf{r}$ is the maximum number of distinct variables in the body of a clause. Hence we can also use the *coarser* but simpler upperbound

$$O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}+max(\mathbf{a},\mathbf{r})}) \tag{6}$$

This upperbound is simpler because it is independent of the order of processing of the literals in clause bodies.

From the previous upperbound (5), we see that much space complexity can often be saved by choosing a processing order for the literals of clauses such that $\mathbf{d}'$ is minimized. A similar remark shall be made concerning time complexity.

5.2 Implementation and Time complexity

A general complexity analysis for arbitrary DLA gives a moderately interesting, though polynomial, result. We shall not consider it here, and we shall restrict our time complexity analysis to the case of the DLA constructed in section 3.2.

However we shall first describe an implementation of our dynamic programming construction that reduces the cost of applying transitions to items.

5.2.1 Implementation

We do not use a direct representation of items. Rather we consider *classes of items* with the same first fact. The class $[\mathrm{F}a]$ of the fact $\mathrm{F}a$ is implemented by a pair composed of

- the fact $\mathrm{F}a$ itself, and
- the set of (pointers to) the classes $[\mathrm{G}_i\, b_i]$ of the facts $\mathrm{G}_i\, b_i$ such that $<\mathrm{F}a\ \mathrm{G}_i\, b_i>$ is an item found by the algorithm.

The classes themselves and the elements within classes are accessed (at least theoretically[21]) via multidimensional arrays indexed by predicates and tuple component values. Thus the location of the class $[\mathrm{G}b]$ of a given fact $\mathrm{G}b$ can be accessed in unit time for whatever purpose.

In particular, we have a constant cost for each attempt to store — it may be there already — a newly produced item in its proper item class.

We also link the elements of each class into a list, so as to access all of them for processing with a total access time proportional to the cardinality of the set, independently of the sparsity of the above arrays. Hence

[18] Evaluating the complexity with respect to the cardinality of $\mathcal{D}_\mathcal{I}$, rather than the number of extensional tuples, gives the more pessimistic result. Indeed the number of tuples exceeds that of the values occurring in them, up to a multiplicative constant (the maximum length of a tuple). In addition, the values not occurring in extensional tuples add to the complexity evaluation, though they will never actually be used in query evaluations.

However, we can always read the parameter $|\mathcal{D}_\mathcal{I}|$ in the complexity formulae as the number of *relevant* values, i.e. those values that are actually used in the computation. Though this definition is too dependent on the algorithm it purports to measure, it at least excludes the values not appearing in extensional tuples of the database (except in case of head variables not appearing in the body of the clause, which are unusual and could probably be economically processed by other means anyway).

[19] We do not detail the role of the initial and final predicates, which do not change the analysis significantly.

[20] These bindings serve for *joins* between body literals, and to bind the variables of the head literal.

[21] Implementations may use various tricks to avoid the cost of these arrays in most cases. The analysis of the time complexity of the algorithm gives some hints as to how the sets of classes could be more efficiently implemented, at least by reducing the number of indices.

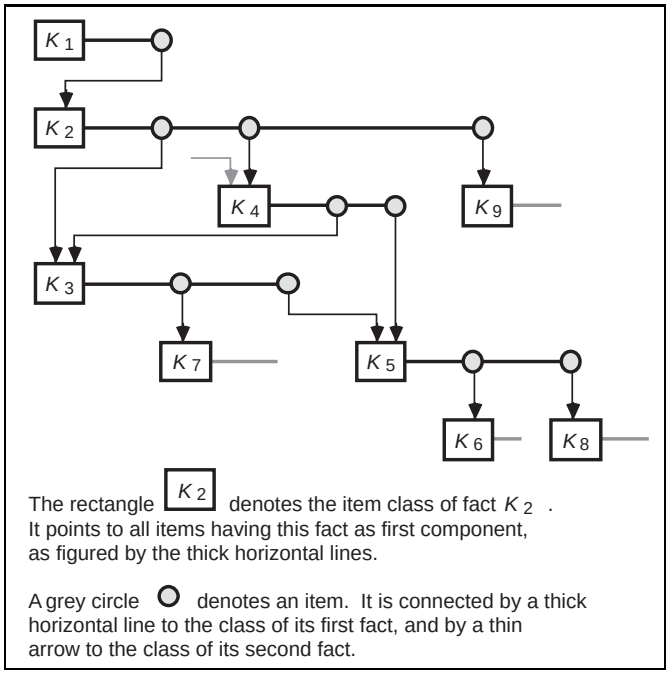


Figure 3: Organization of items into classes.

the access cost is constant when distributed over the elements of the set.

Each class is actually a set of pointers to (the representation of) other classes, so that the class representations are shared and never duplicated. A pointer to the class $[\mathrm{G}_i\, b_i]$ occurring in the class $[\mathrm{F}a]$ may be seen as the representation of the item $<\mathrm{F}a\ \mathrm{G}_i\, b_i>$.

In figure 3, we have represented the items of figure 2 organized into classes as described above. Each rectangle denotes the header of the structure representing the the class of the fact labeling this rectangle. The adjoining thick horizontal line figures the list of items contained in the class. Each circle is a pointer to another class and figures one item. The accessing multidimensional arrays are not represented.

The above description is only a rough sketch, with many details omitted. In particular, an additional structure is needed in each class to keep track of past pop transitions that needed its items as second item (cf. section 4.2.4). These pop transitions still have to be applied to new items later added to the class.

5.2.2 Cost of item production

We already know that, with our implementation, attempting to store a newly obtained item has constant cost, independently of whether the item is actually new or is found to be a duplicate. The same is true when accessing items to apply transitions, with a careful book-keeping to keep track of the items still to be processed (a similar book-keeping is used in the context-free parsing case).

Hence the time complexity is given by the number of items produced, whether new or duplicates, i.e. the number of applications of transitions to items, or (as we shall see) to item classes for push transitions. We first analyze how items are produced by the three kinds of transitions, and then we determine an upperbound for the time complexity of our algorithm.

When applying push transitions, the second fact of items is irrelevant. Thus a push transition need only be applied once to an item class $[\mathrm{F}a]$ independently of the (non-zero) number of items $<\mathrm{F}a\ \mathrm{G}b>$ it contains now or ever. Thus the production of the resulting item $<\nu(\mathrm{H}v)\ \mathrm{F}a>$ has a constant computational cost, for one push transition applied to one item class, and for one extensional fact scanned if it is a scanning transition. The total computational cost of an item imputable to push transitions is proportional to the number of times it can be produced in a different way by applying a push transition to an *item class* and possibly to an extensional fact (for a scanning transition).

A horizontal transition applied to a class $[\mathrm{F}a]$ produces a (usually) distinct item in constant time for each item $<\mathrm{F}a\ \mathrm{G}b>$ in the class. Thus the computational cost of an item imputable to horizontal transitions is proportional to the number of times this item is obtained by applying a horizontal transition to an other item. If the transition is scanning, each application with a different extensional fact is counted separately.

Similarly, the computational cost of an item imputable to pop transitions is proportional to the number of times this item may be obtained by applying this transition to a pair of items $<\mathrm{F}a\ \mathrm{G}b>$ and $<\mathrm{G}b\ \mathrm{M}c>$.

In this analysis we have neglected the cost of scanning extensional tuples. In practice this of course depends on the actual database implementation. From a purely theoretical point of view, we may assume unit cost for each scan by implementing the relations both as linked list of tuples (to have iteration costs proportional to the actual number of tuples in a relation) and as arrays of pointers into these lists (to find a given tuple in unit time).

5.2.3 Time complexity for the bottom-up DLA

We now analyze the time complexity for the bottom-up DLA of section 3.2 by considering the computational cost of the various kinds of items produced by its dynamic programming interpretation.

We recapitulate in figure 4 the definition of the parameters used in the complexity analysis.

An item $<K\ L>$, where K is a $\hat{\mathbf{\Sigma}}$ fact $\hat{\mathrm{A}}b$, can be produced only once by only one type-(1)[22] scanning push

[22]The transition types here refer to the four types of transitions defined in section 3.2 for the construction of the bottom-up DLA.

$\|\mathcal{D}_\mathcal{I}\|$	size of the database as the number of distinct values in the interpretation domain.
$\mathbf{n}$	number of clauses in the Datalog grammar, i.e. $\|\mathbf{\Pi}\|$.
$\mathbf{s}$	maximum arity of extensional predicates.
$\mathbf{v}$	maximum arity of intensional predicates.
$\mathbf{a}$	maximum arity of intensional or extensional predicates, i.e. $max(\mathbf{s}, \mathbf{v})$.
$\mathbf{d}'$	maximum arity of position predicates $\bigtriangledown_{k,i}$ in Pos.
$\mathbf{d}$	maximum arity of stack predicates, i.e. $max(\mathbf{a}, \mathbf{d}')$.
$\mathbf{r}$	maximum number of distinct variables occurring in the body of a clause.
$\mathbf{p}$	a constant such that $\mathbf{p} \leq min(\mathbf{r}, (\mathbf{d}' + \mathbf{a}))$.

Figure 4: Parameters used in the complexity analysis.

transition applied to the item class $[L]$, and to the extensional fact $\text{A}b$. Thus it has unit cost. The total cost for such items is equal to their number, i.e. at most $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{s}+\mathbf{a}})$, since the predicate of L must be in $\mathbf{\Sigma} \cup \mathbf{V}$.

Similarly, an item $<K\ L>$, where K is a $\bigtriangledown_{k,n_k}$ nullary fact, can be produced only once by only one type-(2) non-scanning push transition applied to the item class $[L]$. Thus it has unit cost and the total cost for such items is equal to their number, i.e. at most $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}})$, since the predicate of L must be in $\mathbf{\Sigma} \cup \mathbf{V}$, and K is a nullary fact.

An item $<K\ L>$, where K is a $\hat{\mathbf{V}}$ fact $\hat{\text{F}}a$, can be produced at most a finite number of times by distinct type-(4) horizontal transitions. Precisely there may be as many such transitions — each producing the item at most once — as there are clauses in the original Datalog grammar with an F predicate for the head literal. For each such clause γ_k, there is at most one type-(4) transition ($\bigtriangledown_{k,0} x_{k,0} \diamond \mapsto \text{F}_{k,0} t_{k,0}$) that can produce the item $<K\ L>$ from an item $<K'\ L>$, and K' must be $\bigtriangledown_{k,0} a'$ where a' is a fully determined subtuple of a, which implies the unicity of K' for given K and γ_k. Hence K is uniquely produced for a given γ_k. Thus the time cost of $<K\ L>$ is bounded by a constant proportional to the maximum number of clauses with the same head predicate[23]. Hence, the computational cost of these items is again of the order of their number, i.e. it is at most $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{v}+\mathbf{a}})$.

Finally we consider the only remaining case of an item $<K\ L>$, where $K = \bigtriangledown_{k,i-1} a$ for $i \in [1, n_k]$. Such an item can only be produced by the unique type-(3) transition ($\bigtriangledown_{k,i} x_{k,i}\ \hat{\text{F}}_{k,i} t_{k,i} \mapsto \bigtriangledown_{k,i-1} x_{k,i-1}$) from two other items $<\bigtriangledown_{k,i} b\ \hat{\text{F}}_{k,i} c>$ and $<\hat{\text{F}}_{k,i} c\ L>$. From the definition of the tuples of variables $x_{k,i}$ used with position predicate $\bigtriangledown_{k,i}$, we know that $x_{k,i-1} \subseteq x_{k,i} \cup \mathbf{X}(t_{k,i})$. Hence the maximum number of item pairs to which this transition is applicable is bound by the maximum number of different combinations of three facts $\bigtriangledown_{k,i} b$, $\hat{\text{F}}_{k,i} c$ and L to which the transition may be applied. Thus the computational cost of such items is at most $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{d}'+\mathbf{a}+\mathbf{a}})$, since there is only a finite number of type-3) transitions (precisely the total number of clause body literals).

However, we must note that in the above discussion, the variables in $x_{k,i} \cup \mathbf{X}(t_{k,i})$ all come from the same clause body of the original Datalog grammar since $i \in [1, n_k]$, and also that these variable are the only degrees of freedom in the choice of the fact $\hat{\text{F}}_{k,i} c$ which must match $\hat{\text{F}}_{k,i} t_{k,i}$. Hence we can replace the first two terms of the exponent by the maximum number $\mathbf{r}$ of variables in a clause body, if it happens to be smaller. The computational cost of these items is then

$$O(|\mathcal{D}_\mathcal{I}|^{\mathbf{p}+\mathbf{a}})$$

where $\mathbf{p} = min(\mathbf{r}, (\mathbf{d}' + \mathbf{a}))$. In fact we could even give a finer definition of $\mathbf{p}$ as the maximum cardinality of $x_{i,k} \cup \mathbf{X}(t_{i,k})$ for $k \in [1, \mathbf{n}]$ and $i \in [1, n_k]$, i.e. the maximum number of variables in a body literal added to the number of bindings transiting through that literal when using the clause to prove an instance of the head.

Since $\mathbf{a} = max(\mathbf{s}, \mathbf{v})$ a common upperbound for the first three kinds of items is $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{2a}})$. Hence, for all items we have an upperbound:

$$O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}+max(\mathbf{p},\mathbf{a})}) \tag{7}$$

From the inequality $\mathbf{p} \leq \min(\mathbf{r}, \mathbf{d}' + \mathbf{a})$ we get $max(\mathbf{p}, \mathbf{a}) \leq max(\mathbf{r}, \mathbf{a})$ and hence the coarser time complexity upperbound

$$O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}+max(\mathbf{a},\mathbf{r})}) \tag{8}$$

which is the same as the coarse space upperbound (6).

Thus the same remarks apply for time and space complexity upperbounds, concerning independence of the coarser bound from the processing order of literals, and concerning the optimization of the algorithm by a proper choice of this processing order.

[23] A tighter value is: the maximum number of clauses with *unifiable* heads.

6 Comments and Comparison to other work

The use of stateless automata was motivated by

- a simplification of the presentation, though all the results presented here could be adapted to a DLA with facts as states,
- a better complexity analysis, since careless use of states could increase the computational complexity of the dynamic programming interpretation.

However, it could be sometimes useful to introduce a finite set of state symbols (not predicates), so as to simplify the presentation of theoretical constructions or of optimizations. This would have no impact on the degree of the polynomial complexity.

In this paper we chose to keep the DLA construction extremely simple to avoid confusing issues, and to see how much can be gained by the dynamic programming strategy alone. As presented in section 3.2, the DLA construction implicitly fixes the way each clause is being used in the query processing: literals are always processed in the same order independently of the already known bindings. This is certainly restrictive, but it is quite possible to use more sophisticated constructions akin to *magic sets*, *sideways information passing* and others [2, 34, 14], that take into account the already established variable bindings to decide the order in which the remaining literals of a clause instance ought to be processed. Such constructions correspond either to top-down (i.e. backward-chaining) query evaluation, or to bottom-up evaluation usually optimized with a *predictive* top-down components somewhat similar to the predictive feature of some bottom-up context-free parser (e.g. LR(k) parsers). Top-down and bottom-up evaluation seems to correspond to the two ways of passing information (cf. [3]), respectively by unification with rule head and by sideways passing.

The restating of existing optimized query evaluation techniques as DLA constructions has yet to be investigated. However, though we expect from it some drastic complexity improvements for specific examples, or for subclasses of Datalog programs, it is not clear whether and how these optimizations will improve the above complexity upperbounds in the general case.

Our coarse time and space complexity upperbound $O(|\mathcal{D}_\mathcal{I}|^{\mathbf{a}+max(\mathbf{a},\mathbf{r})})$ is similar to the upperbound given by Vieille in [38, 37], i.e. $O(|\mathcal{D}_\mathcal{I}|_{\text{rel}}^{\mathbf{v}+k'})$ where $|\mathcal{D}_\mathcal{I}|_{\text{rel}}$ is the number of relevant individuals of the database[24], and $k' \leq \mathbf{v} + \mathbf{k}$, $\mathbf{k}$ being the maximum number of variables in a clause body that do not appear in the head (hence $max(\mathbf{a}, \mathbf{r}) \leq \mathbf{v}+\mathbf{k} = \mathbf{k}')$[25]. Veille's results are themselves refinements of earlier theoretical results by Immerman [11] giving the complexity upperbound $O(|\mathcal{D}_\mathcal{I}|^{2^{\mathbf{v}+\mathbf{k}}})$ in a somewhat different setting.

We get this result with a rather straightforward general construction, to which it should be easy to add specialized optimizations (see above) without having to reestablish complete proofs of soundness and completeness. This is achieved by expressing the optimizations in terms of the original non-deterministic DLA, and relying on the general results concerning the soundness and completeness of the dynamic programming interpretation.

Comparison with many of the algorithms otherwise proposed in the literature is difficult, since they are not often accompanied by an analysis of their complexity or by implementation details, and are sometimes aiming at different computational architectures (e.g. parallel or distributed). Precise relation to the work of Vieille is not completely clear to us at this point. It seems that the construction of the DLA may keep a non-deterministic choice in the processing order of the literals of a clause, which means that we do not require a "*homogeneous selection function*" [38]. At worst, the effect of such a policy would be to have in the complexity results a constant that is an exponential function of the maximum number of literals in a clause body. However the DLA construction implicitly enforces *locality* of the selection function as defined in [36].

The pushdown DLA approach seems a good computational model on sequential processors, though it probably restricts a bit the ways in which the computation may be performed [30]. However there is no evidence that this constraint may cause the computational performances to be reduced. The approach can handle top-down as well as bottom-up query evaluation, with polynomial results, depending on the way the DLA is constructed. Finally, the order in which the items are constructed and used has not been fixed here, and can be chosen to be depth-first or breadth-first, though depth-first is probably preferable to optimize away useless computations when only one answer fact is needed, either for a query or for a subquery.

The complexity upperbounds obtained are somewhat coarse, and can probably be refined in many special cases. Furthermore past experience with similar algorithms in the case of context-free parsing shows that the expected cubic complexity is rarely observed, and that the parsing algorithm often behaves linearly by somehow adapting to the real complexity of the problem at hand. Hopefully this optimistic observation will extend to the case of most recursive queries.

[24] Vieille's definition of "*relevant individuals of the database*" may not be the same as ours.

[25] We neglect here the difference between $\mathbf{a}$, i.e. $\mathbf{s} + \mathbf{v}$, and $\mathbf{v}$ because the extensional tuples are preexisting to the query computation and are rather part of the input. Thus it is not really fair to count them as additional complexity.

Acknowledgements: Although the results reported here were developed independently of those of Laurent Vieille, the initial motivation for this research came from his presentation of an overview of his work [39], while the author's current research interest was on extending the dynamic programming parsing techniques presented in [16]. The author acknowledges with pleasure the time L. Vieille spent explaining to him the basic problem of recursive query processing, as well as the moral and technical support (and patience) of the INRIA database and Prolog community, and particularly Serge Abiteboul and Michel Scholl.

The referees are also thanked for their constructive criticisms and suggestions.

References

[1] Aho, A.V.; and Ullman, J.D. 1972 *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey.

[2] Bancilhon, F.; and Ramakrishnan, R. 1986 An Amateur's Introduction to Recursive Query Processing Strategies. *Proc. of ACM Internat. Conf on Management of Data, SIGMOD'86* : 16-52, Washington (D.C.).

[3] Beeri, C.; and Ramakrishnan, R. 1987 On the Power of Magic. *Proc. of the* 6^{th} *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*: 269-283, San-Diego (California).

[4] Bouckaert, M.; Pirotte, A.; and Snelling, M. 1975 Efficient Parsing Algorithms for General Context-Free Grammars. *Information Sciences* 8(1): 1-26

[5] DeRemer, F.L. 1971 Simple LR(k) Grammars. *Communications ACM* 14(7): 453-460.

[6] Dietrich, S.W.; and Warren, D.S. 1985 *Dynamic Programming Strategies for the Evaluation of Recursive Queries*. Tech. Rep. 85-31, Dept. of Computer Science, SUNY at Stony Brook (New York).

[7] Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 13(2): 94-102.

[8] Graham, S.L.; Harrison, M.A.; and Ruzzo W.L. 1980 An Improved Context-Free Recognizer. *ACM Transactions on Programming Languages and Systems* 2(3): 415-462.

[9] Griffiths, I.; and Petrick, S. 1965 On the Relative Efficiencies of Context-Free Grammar Recognizers. *Communications ACM* 8(5): 289-300.

[10] Hays, D.G. 1962 Automatic Language-Data Processing. In *Computer Applications in the Behavioral Sciences*, (H. Borko ed.), Prentice-Hall, pp. 394-423.

[11] Immerman, N. 1982 Relational Queries Computable in Polynomial Time. *Proc. of the 14th ACM/SIGACT Symposium* :147-152.

[12] Kasami, J. 1965 *An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages*. Report of Univ. of Hawaii, also AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford (Massachusetts), also 1966, University of Illinois Coordinated Science Lab. Report, No. R-257.

[13] Kay, M. 1980 Algorithm Schemata and Data Structures in Syntactic Processing. *Proceedings of the Nobel Symposium on Text Processing*, Gothenburg.

[14] Kifer, M.; and Lozinskii, E.L. 1985 *Query Optimization in Logical Databases*. Tech. Report 85/16, Dept. of Computer Science, State University of New York at Stony Brook, Long Island (New York).

[15] Knuth, D.E. 1965 On the Translation of Languages from Left to Right. *Information and Control*, 8: 607-639.

[16] Lang, B. 1974 Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the* 2^{nd} *Colloquium on Automata, Languages and Programming*, J. Loeckx (ed.), Saarbrücken, Springer Lecture Notes in Computer Science 14: 255-269.
Also: Rapport de Recherche 72, IRIA-Laboria, Rocquencourt (France).

[17] Lang, B. 1988 Parsing Incomplete Sentences. To appear in *Proc. of the* 12^{th} *Internat. Conf. on Computational Linguistics (COLING'88)*, Budapest (Hungary).

[18] Lang, B. 1988 *Complete Evaluation of Horn Clauses, an Automata Theoretic Approach*. In preparation.

[19] Li, T.; and Chun, H.W. 1987 A Massively Parallel Network-Based Natural Language Parsing System. *Proc. of* 2^{nd} *Int. Conf. on Computers and Applications* Beijing (Peking), : 401-408.

[20] Lloyd, J.W. 1984 *Foundations of Logic Programming*. Springer-Verlag.

[21] McKay, D.; and Shapiro, S. 1981 Using Active Connection Graphs for Reasoning with Recursive Rules. *Proc. of 7th IJCAI* :368-374.

[22] Maier D.; and Warren, D.S. 1988 *Computing with Logic — Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company.

[23] Nakagawa, S. 1987 Spoken Sentence Recognition by Time-Synchronous Parsing Algorithm of Context-Free Grammar. *Proceedings of IEEE-IECE-ASJ International Conference on Acoustics, Speech, and Signal Processing (ICASSP 87*, Dallas (Texas), Vol. 2 : 829-832.

[24] Pereira, F.C.N.; and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis — Asurvey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231-278.

[25] Pereira, F.C.N.; and Warren, D.H.D. 1983 Parsing as Deduction. *Proceedings of the* 21^{st} *Annual Meeting of the Association for Computational Linguistics*: 137-144, Cambridge (Massachusetts).

[26] Phillips, J.D. 1986 A Simple Efficient Parser for Phrase-Structure Grammars. *Quarterly Newsletter of the Soc. for the Study of Artificial Intelligence (AISBQ)* 59: 14-19.

[27] Porter, H.H. 3rd 1986 *Earley Deduction*. Tech. Report CS/E-86-002, Oregon Graduate Center, Beaverton (Oregon).

[28] Pratt, V.R. 1975 LINGOL — A Progress Report. In *Proceedings of the 4th IJCAI*: 422-428.

[29] Robinson, J.A. 1965 A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23-41.

[30] Sheil, B.A. 1976 Observations on Context Free Parsing. in *Statistical Methods in Linguistics*: 71-109, Stockholm (Sweden), Proc. of Internat. Conf. on Computational Linguistics (COLING-76), Ottawa (Canada).
Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard Univ., Cambridge (Massachusetts).

[31] Shieber, S.M. 1985 Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. *Proceedings of the* 23^{rd} *Annual Meeting of the Association for Computational Linguistics*: 145-152.

[32] Tomita, M. 1987 An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[33] Kuniaki Uehara; Ryo Ochitani; Osamu Kakusho; Junichi Toyoda 1984 A Bottom-Up Parser based on Predicate Logic: A Survey of the Formalism and its Implementation Technique. *1984 Internat. Symp. on Logic Programming*, Atlantic City (New Jersey), : 220-227.

[34] Ullman, J.D., 1985 Implementation of Logical Query Languages for Databases. *ACM transactions on Database Systems*, 10(3):289-321.

[35] Van Emden, M.H.; and Kowalski, R.A. 1976 The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM* 23(4): 733-742.

[36] Vieille, L. 1987 Database-Complete Proof Procedures Based on SLD Resolution. *Proc. of the 4th Internat. Conf. on Logic Programming* Melbourne (Australia).

[37] Vieille, L. 1987 *Recursive Query Processing: The power of Logic*. Tech. Report TR-KB-17, European ComputerIndustry Research Center (ECRC), Munich (West Germany).

[38] Vieille, L. 1987 *From QSQ towards QoSaQ: Global Optimization of Recursive Queries*. Tech. Report TR-KB-18, European ComputerIndustry Research Center (ECRC), Munich (West Germany). To appear in *Proc.* 2^{nd} *International Conf. on Expert Database Systems*, April 1988.

[39] Vieille, L. 1987 Recursive Query Processing: Theoretical and Practical Aspects. *France-Japan Artificial Intelligence and Computer Science Symposium* :243-260, Cannes (France).

[40] Younger, D.H. 1967 Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2): 189-208

A Datalog and Context-Free Parsing

In this appendix, we attempt to exhibit informally the relation between *context-free (CF)* parsing and evaluation of recursive queries. We should first remark that the existence of such a relation can naturally be inferred from the well recognized parenthood between recursive queries and Horn clauses on one hand, and the application of Horn clauses to parsing problems by means of Definite Clause Grammars [24, 25] on the other hand.

We shall proceed informally by describing the CF parsing problem in terms of recursive query evaluation. Our approach is similar to that of Pereira and Warren for definite clause grammars.

A.1 Context-free parsing

A CF grammar $\mathbf{G}$ is defined as a 4-tuple:

$$\mathbf{G} = (\mathbf{V}, \mathbf{\Sigma}, \mathbf{\Pi}, \overset{\circ}{\mathrm{N}})$$

where:

- $\mathbf{V}$ is a finite set of nonterminal symbols.
- $\mathbf{\Sigma}$ is a finite set of word symbols.
- $\mathbf{\Pi}$ is a finite set of production (or reduction) rules of the form $\mathrm{N} \rightarrow \eta$ where the *left-hand-side (LHS)* N is a nonterminal in $\mathbf{V}$, and the *right-hand-side (RHS)* η is a finite sequence of symbols in $\mathbf{V} \cup \mathbf{\Sigma}$. There may be several rules with the same LHS.
- $\overset{\circ}{\mathrm{N}}$ is the *initial nonterminal*, or *axiom*.

In figure 5 we give an example of CF grammar, defined only by its rules. By convention we write nonterminal symbols with uppercase letters and terminal symbols (i.e. lexical categories or words[26]) with lowercase letters. The axiom is the LHS of the first rule.

Given a sequence ξ of symbols in $\mathbf{V} \cup \mathbf{\Sigma}$, we say that ξ *reduces immediately* to ξ' by the rule $\rho_k : \mathrm{N}_k \rightarrow \eta_k$ iff there are two sequences ξ_1 and ξ_2 such that $\xi = \xi_1 \eta_k \xi_2$ and $\xi' = \xi_1 \mathrm{N}_k \xi_2$.

We say that ξ_0 *reduces* to ξ_n iff for every index i in the range $[1..n]$ the sequence ξ_{i-1} reduces immediately to ξ_i.

Given an *input sentence* s, i.e. a sequence of symbols from $\mathbf{\Sigma}$, the CF recognition problem consists in attempting to reduce this sentence s to a string containing only the axiom $\overset{\circ}{\mathrm{N}}$ of the CF grammar $\mathbf{G}$. Iff this reduction is possible, the input sentence is said to belong to the language $\mathcal{L}(\mathbf{G})$ defined by the grammar $\mathbf{G}$.

[26] For simplification English words are replaced by their lexical category before parsing.

Grammar:

```
(1)     S   --> NP    VP
(2)     S   --> S     PP
(3)     NP  --> n
(4)     NP  --> det   n
(5)     NP  --> NP    PP
(6)     PP  --> prep  NP
(7)     VP  --> v     NP
```

Input sentence: `John sees a man with a mirror`

Input as lexical categories: `n v det n prep det n`

Two possible reduction sequences represented by the indices of the grammar rules used in the reduction:
`3 4 7 1 4 6 2` and `3 4 4 6 5 7 1`

Figure 5: A CF grammar defining simple English sentences

There are other equivalent definitions of the CF recognition problem (see for example [1]). In practice, people are more interested in CF parsing, i.e. they also want to know the sequence of rules of the grammar used in the reduction process. This sequence is used to structure the input sentence into a *parse-tree*, i.e. a proof-tree, that is used to derive some attached semantics from the syntactic structure. A context-free parser is an algorithm that recognizes the sentences of a CF language defined by its grammar, and produces (at least) one reduction sequence for each recognized sentence. We do not consider this aspect here.

Figure 5 contains an example of a sentence belonging to the language of the given grammar, and two possible reduction sequences for that sentence. The sentence used is a sequence of lexical categories of English, and it corresponds (for example) to the actual English sentence "`John sees a man with a mirror`".

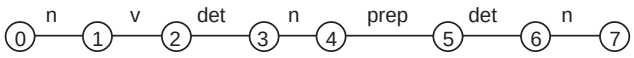
A.2 A Datalog view of context-free parsing

Restating the definite clause grammar approach [24, 25] in database terminology, we may view the input sentence as an extensional database in which the individuals (attribute values) are the positions between words of the input sentence, and the terminal symbols are extensional binary relations between these individuals. It is convenient to name the positions in the input sentence with the integers from 0 to n where n is the size (i.e. number of words) of the input sentence. These integers are only labels and have no arithmetic role. Accordingly, the input sentence of figure 5 may then be understood as the database represented pictorially in figure 6.

Extensional database corresponding to the input sentence of figure 5:

`n v det n prep det n`

`n` = {(0,1), (3,4), (6,7)}
`v` = {(1,2)}
`det` = {(2,3), (5,6)}
`prep` = {(4,5)}



Intensional relations corresponding to the CF grammar given in figure 5:

(1)	S(x,y)	:- NP(x,z)	VP(z,y)
(2)	S(x,y)	:- S(x,z)	PP(z,y)
(3)	NP(x,y)	:- n(x,y)	
(4)	NP(x,y)	:- det(x,z)	n(z,y)
(5)	NP(x,y)	:- NP(x,z)	PP(z,y)
(6)	PP(x,y)	:- prep(x,z)	NP(z,y)
(7)	VP(x,y)	:- v(x,z)	NP(z,y)

Figure 6: A recursive database corresponding to the previous CF example.

The nonterminal symbols of the grammar are then considered as intensional binary relations. These intensional relations are defined by clauses obtained by the following transformation of the rules of the original CF grammar:

For each CF rule

$$\mathrm{N} \longrightarrow \mathrm{z}_1 \ldots \mathrm{z}_p$$

one creates the clause

$$\mathrm{N}(x_0, x_p) \text{ :- } \mathrm{z}_1(x_0, x_1) \ldots \mathrm{z}_p(x_{p-1}, x_p)$$

where the symbols "z_i" are terminals or nonterminals, and the symbols "x_i" are variables.

Thus the CF grammar of figure 5 becomes the set of clauses of figure 6 which define intensional relations on any extensional database corresponding to a sequence of symbols in $\mathbf{\Sigma}$.

An input sentence of length n belongs to the language $\mathcal{L}(\mathbf{G})$ if the recursive query $\overset{\circ}{\mathrm{N}}(0, n)$ can be answered successfully in the corresponding database. In the case of the example of figure 5 and 6, the input sentence belongs to the language of the given grammar since the tuple (0,7) belongs to the extensional relation `S`.

In actual applications of definite clause grammars [24, 25], more complex relations with more than two arguments are usually considered. The first two arguments of these relations correspond to the above description, while the other arguments carry the semantical information to be deduced from the syntactic structures.

Input sentence: `John saw a man with a mirror`

Input as lexical categories: `n n|v det n prep det n`

Extensional database:

`n` = {(0,1), (1,2), (3,4), (6,7)}
`v` = {(1,2)}
`det` = {(2,3), (5,6)}
`prep` = {(4,5)}

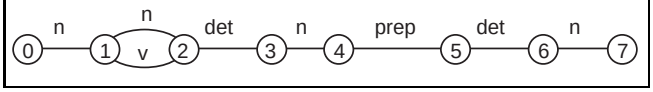


Figure 7: Input database with a multi-part-of-speech word.

A.3 More complex examples

In the above comparison, context-free parsing corresponds to very degenerate Datalog programs: the extensional database is always reduced to a chain. Less degenerate examples may be found in the parsing technology developed for natural language processing. We shall consider three kinds of problems: the so-called "multi-part-of-speech" words, parsing in word lattices, and parsing incomplete sentences.

A.3.1 Multi-part-of-speech words

A *multi-part-of-speech* word is a word that belongs to several lexical categories [32]. In the context of programming languages, such words may be encountered for example when the keywords of the language are not reserved and may also be used as identifiers.

In the context of natural languages this situation occurs frequently because of homonymies. For example the word "`saw`" may be understood as a noun for a tool or as the past tense of a verb. Since the conflict cannot be resolved without using the context, one must consider both possibilities when attempting to parse the sentence "`John saw a man with a mirror`". This is reflected in our database view by having an extensional database which is no longer a chain, as shown in figure 7.

Note that our labeling of places between words is not sufficient to keep track of the interpretation of the word "`saw`" used in a successful parse. This may be done by other techniques, including the use of non binary relations.

A special case of parsing with multi-part-of-speech words is the parsing of a sentence containing an unknown word [32]. Such a word must then be considered a priori as a multi-part-of-speech word that belongs to all lexical categories, at least until parsing eliminate some possibilities.

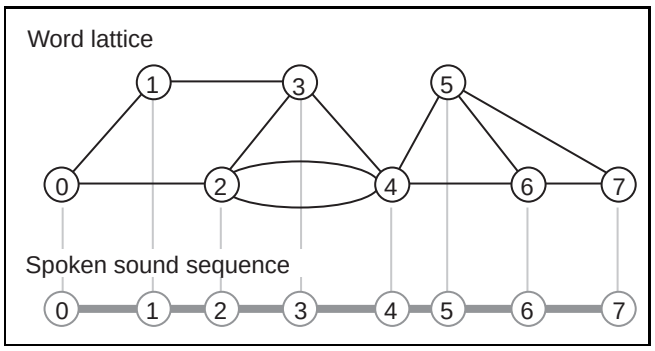


Figure 8: Input database for a word lattice.

A.3.2 Parsing in a word lattice

Parsing in a word lattice is a problem that occurs when attempting to parse spoken sentences [23]. The input sentence is then a continuous sound sequence with no obvious separation between lexical constituents. This input sentence is figured by the thick grey line in figure 8.

It is sometimes the case that this sentence can be decomposed in several different ways to form a sequence of spoken lexical elements. All possible decompositions are then conveniently represented by a partially ordered graph (loosely called *lattice* in the literature, though it need not be one in the mathematical sense of the word). The nodes of this graph correspond to time instants that are *possible* places between words. Each edge represents a word that could correspond to the sound sequence occurring between the time instants corresponding to its end nodes.

Figure 8 contains only an abstract example, without reference to a concrete sentence[27]. The absence of an edge between nodes 0 and 3, 1 and 2, or 3 and 5 (for example) in the lattice indicates that the corresponding segments of the sound sequence cannot be understood as words.

The words lattice thus constructed can then be considered as the extensional database against which the "parsing query" is to be evaluated. The parsing query is as before composed by taking the CF grammar axiom as binary predicate with the two end points of the lattice as arguments.

A.3.3 Parsing incomplete sentences

In all the previous examples, the individuals of the extensional database corresponding to the input sentence may be seen as partially ordered. In this example we introduce simple (one node) cycles[28].

[27]The interested reader can try to build a word lattice for the following classical French holorhyme verses:

Gal, amant de la reine, alla, tour magnanime,
Galamment de l'arène à la tour Magne, à Nîmes.

[28]The author does not know of any other language parsing example where more complex cycles would be meaningful.

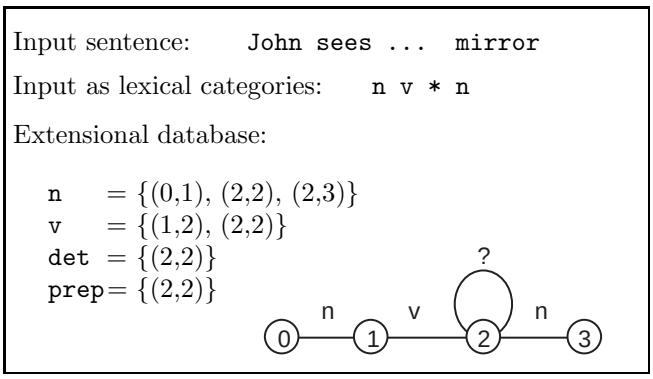


Figure 9: Input database for an incomplete input sentence.

Parsing incomplete sentences corresponds to a situation when part of the input sentence has been lost or deliberately omitted [17]. This case is different from that of the unknown word (see above) because an arbitrary number of words could be in the missing fragment of the sentence (we assume only complete words are missing).

It is then convenient to merge the beginning and the end of the unknown sentence fragment into a unique sentence position, and then have an arc labeled with the unknown word cycling on that position. This give us an elementary cycle in the extensional database corresponding to the input. More precisely, if the position of the unknown fragment is k, the tuple (k,k) must be added to all extensional relations.

The example of figure 9 uses the symbol "`*`" to represent an unknown sentence fragment, and the symbol "`?`" to represent a unique unknown word occurrence. In the extensional database this corresponds to the addition of the tuple (2,2) to all extensional relations.