Complete evaluation of Horn clauses : an automata theoretic approach

Bernard Lang

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 913

Programme 1

COMPLETE EVALUATION OF HORN CLAUSES : AN AUTOMATA THEORETIC APPROACH

Bernard LANG

Novembre 1988

Complete Evaluation of Horn Clauses: an Automata Theoretic Approach

Bernard Lang

Abstract

We show that the execution strategies of Definite Clause Programs, — including resolution in Horn Clause Logic — can be implemented by Push-Down Automata (PDA) that store atoms and substitutions on their stack. By extending to these automata a dynamic programming technique developed for PDAs in context-free parsing, we obtain a general technique for constructing efficient and complete Definite Clause Programs compilers. The use of dynamic programming gives an exponential reduction in complexity over the traditional resolution techniques. This approach can combine indifferently (any variant of) forward or backward chaining with (most variants of) breadth-first or depth-first evaluation, without losing answer completeness. Detailed comparison is made with other published algorithms (Earley deduction, OLDT and SLD-AL resolution), and it is argued that the dynamic programming aspects of earlier proposals may be expressed within the general framework of our formalism. This leads to the conjecture that the LIFO discipline which characterizes PDAs is at the heart of these dynamic programming constructions. Our constructions are formally defined, and correctness proofs are sketched. Search fairness and admissibility tests are analyzed. The results produced by a computer implementation are given as example.

Evaluation Complète des Clauses de Horn: une Approche par la Théorie des Automates

Bernard Lang

Résumé

Nous montrons que les diverses stratégies d'évaluation des programmes de Clauses Définies — en particulier l'utilisation de la résolution pour les Clauses de Horn — peuvent être exprimées par des Automates à Pile (PDA) qui mémorisent des atomes et des substitutions sur leur pile. Nous étendons à ces automates une technique d'évaluation par programmation dynamique originellement développée pour les PDAs dans le cadre de l'analyse syntaxique des langages sans contexte. Nous obtenons ainsi une technique générale pour construire des compilateurs efficaces et complets pour les programmes de clauses définies. L'utilisation de la programmation dynamique donne une réduction exponentielle de la complexité des calculs par rapport aux techniques traditionelles fondées sur la résolution. Cette méthode peut combiner (toute variante de) chaînage avant ou arrière avec (la plupart des variantes de) l'évaluation en largeur ou en profondeur d'abord, sans perdre la complétude des réponses. Nous faisons une comparaison détaillée avec d'autres algorithmes publiés ("Earley deduction", résolution OLDT et SLD-AL) et nous montrons que ces méthodes peuvent être exprimées dans le cadre général de notre formalisme. Nous en conjecturons que la gestion de la mémoire en pile qui caractérise les PDAs joue un rôle essentiel dans ces techniques de programmation dynamique. Les constructions sont définies formellement et les preuves de correction sont esquissées. Nous analysons en particulier les problèmes de recherche équitable des solutions et l'élagage de l'espace de recherche par des tests d'admissibilité. Un exemple produit par une implantation sur ordinateur est donné en appendice.

Complete Evaluation of Horn Clauses: an Automata Theoretic Approach

Bernard Lang
INRIA
B.P. 105, 78153 Le Chesnay, France
e-mail: lang@inria.inria.fr

October 17, 1988

Abstract

We show that the execution strategies of Definite Clause Programs, — including resolution in Horn Clause Logic — can be implemented by Push-Down Automata (PDA) that store atoms and substitutions on their stack. By extending to these automata a dynamic programming technique developed for PDAs in context-free parsing, we obtain a general technique for constructing efficient and complete Definite Clause Programs compilers. The use of dynamic programming gives an exponential reduction in complexity over the traditional resolution techniques. This approach can combine indifferently (any variant of) forward or backward chaining with (most variants of) breadth-first or depth-first evaluation, without losing answer completeness. Detailed comparison is made with other published algorithms (Earley deduction, OLDT and SLD-AL resolution), and it is argued that the dynamic programming aspects of earlier proposals may be expressed within the general framework of our formalism. This leads to the conjecture that the LIFO discipline which characterizes PDAs is at the heart of these dynamic programming constructions. Our constructions are formally defined, and correctness proofs are sketched. Search fairness and admissibility tests are analyzed. The results produced by a computer implementation are given as example.

Key Words: Prolog, Horn Clauses, Definite Clauses, Logic Programming, Complete Computations, Push-Down Automata, Non-Determinism, Dynamic Programming, Earley Deduction, Recursive Queries.

Contents

1 Introduction 1

1.1 The context . . . 1

1.2 Overview of the results . . . 3

1.3 Organization of the paper . . . 5

2 Definite Clause Programs 7

3 Logical Push-Down Automata 8

3.1 Definitions . . . 9

3.1.1 Ground computations . . . 10

3.1.2 General computations . . . 11

3.1.3 Equivalence of ground and general computations . . . 11

3.2 Equivalence of LPDAs and DC programs . . . 12

3.2.1 Reduction of DC programs to LPDAs . . . 13

3.2.2 Construction of a backward chaining LPDA . . . 14

3.2.3 Construction of a forward chaining LPDA . . . 17

3.3 Reduction of LPDAs to DC programs . . . 18

4 Dynamic Programming Execution of LPDA 18

4.1 The data structure . . . 19

4.2 Ground dynamic programming interpretation . . . 20

4.2.1 The algorithm . . . 20

4.2.2 Correctness of the algorithm . . . 21

4.3 General dynamic programming interpretation . . . 24

4.3.1 The algorithm . . . 24

4.3.2 Correctness of the algorithm . . . 26

4.4 Fairness . . . 27

4.5 Admissibility . . . 28

5 Comparison with other algorithms 30

5.1 Overview of the main dynamic programming algorithms . . . 30

5.2 Comparison with Earley deduction . . . 33

5.3 Comparison with Bottom-up resolution 35

6 Conclusion **36**

A Other examples of LPDA construction techniques **45**

A.1 OLDT resolution with zero term-depth 45

A.2 Earley deduction 47

B An Implemented Example **50**

1 Introduction

1.1 The context

The syntactical appearance of Horn Clauses (here called simply *clauses*) is very similar to that of *context-free (CF)* grammars. Indeed the Prolog language that popularized them was initially intended (among other purposes) as a generalization of CF grammars, which had proved insufficient to represent the syntactic structure of natural languages [Col-78,Coh-88]. Since then, generally under the name of *definite clauses*, they have been largely used as such a generalization [PerW-80, PerW-83,Shi-85][1]. However, and surprisingly, there seems to have been few attempts to build on the similarity between the two formalisms so as to generalize to Horn clauses some of the theoretical and technical results obtained in the realm of CF languages, grammars and parsing, and specifically the use of *automata* and the corresponding form of non-deterministic reasoning. Still, some of the results established for CF languages have indeed been extended to Horn clauses. Among these, and of specific interest here, the general CF parsing algorithm due to Earley was extended originally by Pereira and Warren [PerW-83], and studied by other authors [Por-86].

Strategies for executing Horn-clause programs, (here called *Definite Clause Programs or DC programs*), are still often expressed directly as symbolic manipulations of the program goal and clauses, corresponding usually to some variant of resolution [Rob-65,Llo-87], though considerable work has been done to optimize these manipulations, especially in the restricted case of Datalog [BanR-86,Ull-85]. On the other hand, CF parsing is based on the compilation of CF grammars into efficient code for a specialized parsing engine: the *Push-Down Automaton (PDA).*

We propose here to define a formal logical engine for the execution of DC programs called the *Logical Push-Down Automaton (LPDA).* A LPDA is essentially a *non-deterministic* PDA that stores logical atoms and substitutions on its stack, and uses unification to apply its transition. We are thus following a well established methodology for computational paradigms (e.g. context-free parsing, functional programming) by associating to a denotational formalism an equivalent and closely related operational one. The former is more tractable for analyzing formal properties of programs, while the latter is the basis for sound and efficient implementations.

We show that a DC program can indeed be compiled into a LPDA by giving as examples some

[1] CF grammar may be seen as sets of ordered Horn clauses where all predicates are binary and all clauses have the so-called *chain property.*

simple LPDA construction techniques. The LPDAs thus obtained are non-deterministic. Naive simulation of their non-deterministic computations may not be very efficient[2], and it has to be performed in breadth-first mode if completeness is desired.

Complete and more efficient simulation of their computation may be obtained by generalizing to LPDAs a dynamic programming technique[3] originally developed by the author for standard non-deterministic PDAs [Lan-74,Lan-88a] as a generalization of Earley's algorithm [Ear-70] for Context-Free parsing[4]. The intent was to combine the wealth of efficient PDA construction techniques developed for the compiler technology with the mechanism proposed by Earley to handle non-determinism[5], and thus obtain efficient general context-free parsers. The idea was later successfully reused by Tomita in the context of natural language parsing [Tom-87]. These earlier results may also be seen as a unifying framework for describing the numerous Earley-like algorithms that were and still are being developed [Lan-71,BouPS-75,Pra-75,Kay-80,Tom-87,Phi-86,Voi-88].

The work presented here follows the same path in the more general setting of Horn clauses. Hence it is technically close to the Earley deduction algorithm proposed by Pereira and Warren [PerW-83]. However, as in the CF case, it has a simpler structure, and it combines more easily with the various techniques developed to optimize the interpretation of DC programs.

The formalisms and algorithms investigated here have applications in several areas: parsing of programming and natural languages, recursive databases and logic programming. To our knowl-

---

[2]However we believe that it may be as efficiently implemented as Prolog, since our PDA model seems fully compatible with the evaluation strategy of modern Prolog compilers [MaiW-88].

[3]This technique has been classified as dynamic programming in earlier literature, though it is a very simple form of it. It is essentially similar to the memo-functions used in functional programming [Bir-80] as was noticed by Tamaki and Sato [TamS-86].

[4]Another dynamic programming construction for non-deterministic PDAs was published earlier by Aho, Hopcroft and Ullman [AhoHU-68]. However it is a tabular algorithm similar in structure to the earlier CYK algorithm for CF parsing [Hay-62,Kas-65,You-66]. The main drawback of these algorithms is that they always require their worst complexity, contrary to Earley type algorithms that often behave much more efficiently than indicated by the worst case complexity analysis.

[5]Its mix of top-down and bottom-up evaluation is often presented as an essential characteristic of Earley's algorithm. We believe it to be rather a historical accident related to the contemporary research on LR(k) parsing [Knu-65]. Our PDA based approach was our way of stripping the dynamic programming construction down to its bare essentials. Sheil proposes in [She-76] a slightly more general construction that abandons the left-to-right structure of Earley's algorithm. This generalization can be expressed within the LPDA formalism.

edge, the three published dynamic programming algorithms for Horn clauses actually come from these three areas:

- the work of Vieille aims at developing general and efficient strategies (QoSaQ) for recursive database queries[Vie-87c],
- the OLDT resolution algorithm of Tamaki and Sato was explicitly intended for logic programming applications [TamS-86], and
- the Earley deduction algorithm of Pereira and Warren [PerW-83] was intended for natural language parsing, since unification based formalisms have gained considerable importance in this field [KapB-82,Shi-84,Kay-84].

The original context-free algorithm of Earley [Ear-70] was intended for both programming and natural languages. The present author's early work [Lan-71,Lan-74] was also intended for syntactically extensible programming languages. The more general logic based techniques are now gaining interest in the programming language community: in particular, it allows syntax extensions depending on the type context of program fragments by combining parsing and type checking into a single process [AasPS-88].

The applications of our techniques are by no means limited to the areas mentioned above. For example, Horn clauses can play an important role in the semantics of programming languages. This is evidenced by the actual implementation of systems based on "natural semantics" [Kah-87,Des-88], and has been justified on more theoretical grounds by Makowsky in [Mak-87].

1.2 Overview of the results

The use of dynamic programming has already been considered by several authors both in the context of recursive queries in Datalog [Vie-87c,RoeLK-86,Lan-88b,SekI-88,KemT-88] and in the context of Horn clauses [PerW-83,TamS-86,Por-86,Die-87,Vie-87a]. In these cases, as well as in Context-Free parsing, *the advantage of our approach is that it separates the execution strategy*[6] *and its optimizations, which are embodied in the LPDA construction, from the implementation of the LPDA interpreter and its handling of non-determinism (backtrack vs breadth-first, with or without*

[6]Execution strategies correspond in particular to top-down vs bottom-up in CF parsing [GriP-65,Kay-80], and backward-chaining vs forward chaining in logic programming, but many variants are possible as shown in appendix A.

dynamic programming). This results in simpler analysis and better understanding of dynamic programming evaluators, and in greater applicability of all constructions.

Hence we believe it is inappropriate to consider the introduction of the LPDA as an intermediary step that adds complexity to the construction of dynamic programming interpreters for DC programs. Rather we see it as an interface structure that decomposes the problem into two simpler and independent ones.

In particular the correctness of the dynamic programming interpretation of LPDAs may be proved independently of the way they were constructed. Similarly, new execution strategies or optimizations may be defined and proved correct as techniques for compiling DC programs into LPDAs, independently of the LPDA interpreter to be used.

Correctness proofs are based on extensional semantics of DC programs as presented in [VanK-76, Llo-87], i.e. a logical formalism is characterized by the set of ground atoms it defines, proves, recognizes or computes. Hence we first prove equivalence of our constructions (non-deterministic LPDA, and deterministic dynamic programming algorithm) and resolution semantics using only ground definitions. Then we extend these results to the general non-ground constructions by means of lifting lemmas for completeness and lowering lemmas for soundness.

To our knowledge, only Tamaki and Sato [TamS-86] and Vieille [Vie-87b] have attempted to prove formally the correctness of a dynamic programming interpretation of Horn clauses. An informal proof of Earley deduction [PerW-83] has also been proposed by Porter [Por-86]. We believe that *the proofs presented here are both simpler and more general* than the previous ones. They are simpler both because the LPDA model used gives rise to very simple structures in the dynamic programming construction, and because the proof of the correctness of the LPDA — i.e. the execution strategy — is separated from the correctness proof of the dynamic programming interpretation of the LPDA. They are more general because the dynamic programming construction can be used for any execution strategy, with only the correctness of the LPDA to be proved again. Correctness proof of a LPDA is essentially similar to the (ground) proof of the variant of resolution embodied in the definition of the LPDA.

More importantly, on the basis of existing literature on this topic, *we conjecture that LIFO memory management is at the heart of all dynamic programming interpretation of DC programs (cf. section 5.1). Hence our LPDA formalism is probably the most natural support for studying these techniques,* and for developing a sound and complete logic programming technology as was

advocated by Tamaki and Sato in [TamS-86].

The greater structural simplicity of our approach to the dynamic programming construction gives a better understanding of the computational phenomena and should allow in the long run more efficient implementations of these techniques which have important applications in several areas. This is particularly clear in the case of ground implementations, as detailed in an earlier publication [Lan-88b], where it is shown that *LPDA-based dynamic programming interpreters can achieve the best known general polynomial complexity bounds in finite computations* [Imm-82,Vie-87b,Vie-87c], and can be finely tuned to do better in specific cases.

The techniques presented here have been implemented in a prototype system [VilZ-88]. This system has been used to produce the example given in appendix B. Our current research concerns the independent study of the compilation of Horn clause programs into LPDAs, and of the efficient implementation of the dynamic programming construction for LPDAs.

1.3 Organization of the paper

The formal structure of this paper is given in the diagram of figure 1 that summarizes the main technical results.

From the point of view of the technical approach presented in this paper, the various proofs of this diagram do not have the same importance. For example the general non-deterministic computations of LPDAs play no direct role in defining the dynamic programming evaluation technique proposed in this paper, but they show its relation with traditional Prolog evaluators.

OLD resolution has been chosen as a convenient example of operational semantics for Horn clauses. It happens to be, in its backtrack version, the evaluation strategy used by Prolog systems, viz. a left-to-right refutation of subgoals. Correctness of OLD resolution with respect to the various accepted definitions of the semantics of DC programs is a consequence of the general correctness of SLD resolution [Llo-87]. The equivalence of ground and general OLD resolution is not given, and it can be established easily.

OLD resolution is used here *only* to establish formally that any DC program may be compiled into an equivalent LPDA. Better compilation techniques should be developed for practical purposes, drawing on the already existing body of knowledge for efficient DC program evaluation [Ull-85, BanMSU-85,BanR-86,BeeR-87,ZanS-86]. For every new compilation technique, one need only prove

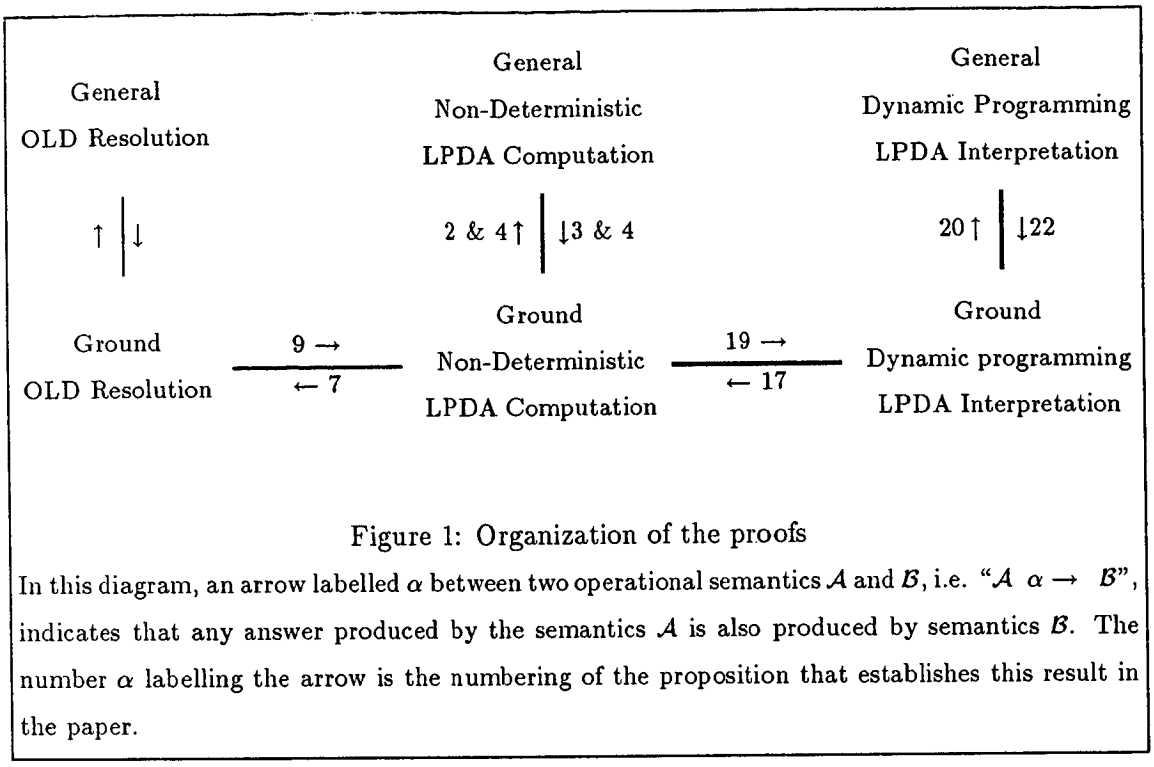

Figure 1: Organization of the proofs

In this diagram, an arrow labelled α between two operational semantics $\mathcal{A}$ and $\mathcal{B}$, i.e. "$\mathcal{A}$ $\alpha \rightarrow$ $\mathcal{B}$", indicates that any answer produced by the semantics $\mathcal{A}$ is also produced by semantics $\mathcal{B}$. The number α labelling the arrow is the numbering of the proposition that establishes this result in the paper.

the correctness of the ground non-deterministic computations of the LPDAs it produces.

All other results, concerning equivalence of various operational interpretations of LPDAs, are independent of the way these LPDAs are produced, and thus apply to any compilation technique. These latter proofs would require adjustment only if we were to modify or add new twists to our interpretations of LPDA.

As previously stated, our proofs are given for ground definitions, and then lifted to general definitions by means of lifting (↑) and lowering (↓) lemmas. Hence we get only extensional equivalence of our operational semantics, i.e. equivalence of the set of ground instances of the answers produced. Intensional equivalence would guarantee that exactly the same set of (usually non-ground) atoms or answer substitutions is produced, but it would require proofs at the general level.

We have chosen not to attempt direct proofs at the general level for several reasons:

- Non-ground proofs are much more intricate since they have to handle more complex substitutions, and the various renamings of variables. In particular, since we shall wish (in the future) to easily prove the correctness of sophisticated compilations techniques from DC programs into LPDAs, we shall want to rely on the much easier ground proofs.
- It isolates the analysis of optimizations specific to general dynamic programming interpre-

tation in the simpler lifting lemmas. This is particularly the case for the admissibility tests analyzed in section 4.5, which can be ignored in the fundamental correctness proofs, that are given in the ground case where admissibility issues are degenerate and thus trivial.

- It is not clear that intensional equivalence can be obtained for all variations that are possible on the basic construction described here, though this has not been much investigated.
- The author does not see any evidence that intensional equivalence is much sought for in practice.

Textually, the paper is organized as follows. Section 2 contains the basic definitions and notations used. In section 3 we define the Logical PDA and show its computational equivalence to Definite Clause Programs. The general dynamic programming construction is defined and studied in section 4. Our approach is compared in section 5 with other published algorithms. More LPDA construction examples are given in appendix A. A complete machine produced example is listed in appendix B.

2 Definite Clause Programs

To emphasize the similarity between DC programs and CF grammars, we follow throughout the paper the presentation of formalisms traditionally used for CF grammars and PDAs.

A DC program $\mathcal{P}$ is a 5-tuple: $\mathcal{P} = (\mathbf{X}, \mathbf{F}, \mathbf{P}, \Gamma, \overset{\circ}{\mathrm{P}})$

where:

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

$\mathbf{F}$ is a finite set of *function* symbols.

$\mathbf{P}$ is a finite set of *predicate* symbols,

Γ is a finite set of $\mathbf{n}+1$ *clauses* γ_k of the form $\mathrm{A}_{k,0}$:- η_k where the *head* $\mathrm{A}_{k,0}$ is an atom, and the *body* η_k is a finite set of atoms $\mathrm{A}_{k,1}, \ldots, \mathrm{A}_{k,n_k}$. There may be several clauses with the same head. Clauses are indexed from 0 to $\mathbf{n}$.

$\overset{\circ}{\mathrm{P}}$ is the *initial predicate.*

We assume without loss of generality that the goal is the atom composed of the initial predicate with only distinct variables as argument, and that the initial predicate appears only as head predicate of the first clause γ_0, i.e. in the atom $A_{0,0}$. This can always be achieved without change in the computed substitutions by adding one clause to the DC program.

We assume known all definitions and results concerning substitutions, subsumption, matching and unification for terms, atoms and pairs or tuples thereof. We also assume familiarity with the basic concepts and results concerning the semantics and the evaluation of DC programs (see for example [VanK-76,Llo-87]).

We note *mgu*(x,y) the *most general unifier* of x and y. The application of substitution σ to A is noted $A\sigma$, and the composition of σ and σ' is noted $\sigma\sigma'$. Subsumption of A by B is noted $A \preceq B$, or $A \preceq_\sigma B$ to emphasize that σ is a substitution such that $B = A\sigma$.

An entity will be qualified as *ground* if it is a structure without variables, or an algorithm that computes only such structures. For convenience, we qualify some entities as *general* to distinguish them explicitly from the ground version, i.e. to emphasize that they can use variables. For clarity, we shall sometimes hat ground entities when they are used together with general ones (e.g. $\hat{A}$), though we do not use this heavier notation when there is no risk of ambiguity.

We note $\mathbf{X}(\heartsuit)$ the set of variables occurring in $\heartsuit$, where the symbol $\heartsuit$ stands for any structured entity (e.g. term, atom, substitution, set of terms, ...). This does not include the substituted variables in substitutions, unless they also appear in some replacement terms. An entity is called a *variant* of another one when they differ only by a renaming of their variables.

3 Logical Push-Down Automata

A LPDA is essentially a PDA that stores atoms and sometimes substitutions in its push-down stack.

A technical point is that we use *stateless* automata. This is achieved without loss of generality by defining popping transitions as replacing the top two atoms of the stack by only one atom. 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 [DeR-71]. In our case, the elimination of states better emphasize the relation between DC programs and LPDAs.

3.1 Definitions

A LPDA $\mathcal{A}$ is defined as a 6-tuple: $\mathcal{A} = (\mathbf{X}, \mathbf{F}, \Delta, \overset{\circ}{\$}, \$_f, \Theta)$

where:

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

$\mathbf{F}$ is a finite set of function symbols[7].

Δ is a finite set of predicate symbols.

$\overset{\circ}{\$}$ is the *initial predicate.* It is a nullary predicate (i.e. a proposition), and constitute the only atom initially on the stack at the start of a computation. It is never removed.

$\$_f$ is the *final predicate.* An atom built with $\$_f$ is called a *final atom.* A final atom is the only atom in the stack, on top of $\overset{\circ}{\$}$ at the end of a successful computation.

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

The transitions in Θ come in three kinds:

horizontal transitions: $B \mapsto C$

push transitions: $B \mapsto CB$

pop transitions: $BD \mapsto C$

where: B, C and D are Δ-atoms, i.e. atoms built with Δ, $\mathbf{F}$ and $\mathbf{X}$.

The initial proposition $\overset{\circ}{\$}$ can only appear in *initial push transitions* of the form: $\overset{\circ}{\$} \mapsto C\ \overset{\circ}{\$}$, and thus it is only an end marker that is never used in any computation after the first step.

Given a LPDA, we can define two types of computations: ground and general. Though ground computations may be viewed as a special case of general computations, it is convenient to define them separately to benefit from the simplicity permitted by the exclusive use of ground atoms.

For both kinds of computations, a *configuration* of the LPDA is composed only of the stack contents. Approximately, the applicability of a transition depends on the first and possibly second atoms stored on the stack, and applying it results in replacing this (these) atom(s) by one or two

[7] Constants are just argument-less functions and require no special treatment.

new ones. This is detailed below. There are usually several transitions that may be applied to a given configuration; hence the LPDA is a non-deterministic automaton[8]. A stack is represented as the sequence of its constituents, with the *stack top on the left-hand side.*

An *initial configuration* is a stack containing only the initial proposition $\overset{\circ}{\$}$. A *final configuration* is a stack containing only a final atom (paired with a substitution in the non-ground case), on top of the initial proposition. There could be several final predicates; we choose to have only one so as to simplify the exposition. This causes no loss of generality.

Each computation step of the LPDA is based on the application of a transition as described below for ground and general computations. A single computation step from a stack configuration ξ to a configuration ξ' using transition τ is denoted by $\xi \vdash_{\tau} \xi'$ or simply by $\xi \vdash \xi'$. We note $\xi \vdash^{n} \xi'$ an n steps computation $\xi = \xi_0 \vdash \xi_1 \vdash \cdots \vdash \xi_n = \xi'$, and we note $\vdash^{\star}$ the reflexive transitive closure of $\vdash$. When both ground and general computations are considered at the same time, we distinguish ground computations by hatting the turnstile symbol $\vdash$ as follows: $\hat{\vdash}$.

3.1.1 Ground computations

In ground computations, the stack contains only ground atoms. Initially the only ground atom is the initial proposition $\overset{\circ}{\$}$. A horizontal (resp. push) transition $B \mapsto C$ (resp. $B \mapsto CB$) is applicable to a stack $A\,\xi$ iff there is a ground substitution s, with domain $\mathbf{X}(B, C)$[9], such that $A = Bs$. The resulting stack/configuration is then $Cs\,\xi$ (resp. $Cs\,Bs\,\xi$). A pop transition $BD \mapsto C$ is applicable to a stack $A\,A'\,\xi$ iff there is a ground substitution s with domain $\mathbf{X}(\overset{\circ}{B}, C, D)$, such that $A = Bs$ and $A' = Ds$. The resulting stack is $Cs\,\xi$.

The LPDA may stop succesfully when it reaches a *final configuration*, which corresponds to a stack containing only one final atom (i.e. built with the final predicate $\$_f$) on top of the initial atom $\overset{\circ}{\$}$. This final atom is the *answer of the computation* that produced it.

Since the LPDA is non-deterministic, it may have many different computations, some of which may not terminate or may stop in a non-final state (i.e. *fail*). The *answer of the LPDA* $\mathcal{A}$ is the set of answers of all successfully terminating ground computations[10]. This set is denoted by

---

[8]In the ground case, the choice of the substitution s used to apply a transition (see below) may also be non-deterministic.

[9]The condition on the domain of s is necessary to ensure that Cs is ground.

[10]Recall that we made the hypothesis that the goal is a predicate with only distinct variables as arguments. Thus

$Answer_g(\mathcal{A})$.

3.1.2 General computations

The technical structure of this paper is such that only ground computations of LPDAs are necessary to state and establish our results (cf. our remarks on the structure of proofs in section 1.3). Thus sections 3.1.2 and 3.1.3 may be skipped by the reader. They have been included because they give a better understanding of the nature of LPDAs, and of the meaning of the structures used later on in general dynamic programming interpretations of LPDAs. Also their description could serve to develop Prolog technology LPDA interpreters.

In general computations, the stack contains pairs A.u composed of a general atom A and a substitution u. The role of the substitution is to memorize bindings that were obtained when computing the atom, and that have not yet been propagated to the lower levels of the stack (since only the top of the push-down stack can be involved in any form of computation). A ground stack is just a degenerate case where all stack substitutions are the identity substitution.

When applying transitions, we now need to use unification instead of simple matching. Like clauses in resolution, each transition has its variables renamed before it is considered for a general computation step of the LPDA. A horizontal (resp. push) transition B $\mapsto$ C (resp. B $\mapsto$ CB) is applicable to a stack A.$u\,\xi$ iff there is a substitution s=mgu(A,B). The resulting stack/configuration is then C$s.us\,\xi$ (resp. C$s.s$ B.$u\,\xi$). A pop transition BD $\mapsto$ C is applicable to a stack A.u A$'$.$u'\,\xi$ iff there is a substitution s=mgu(<A,A$'u$>,<B,D>) The resulting stack is C$s.u'us\,\xi$.

Definitions of initial and final configurations are similar to the ground case with stack pairs containing identity substitutions. The definition of the answer of a computation is also as above. The set of all general answers of the LPDA $\mathcal{A}$ is denoted by $Answer_G(\mathcal{A})$.

3.1.3 Equivalence of ground and general computations

The following definition establishes a subsumption relation between general and ground stacks, which will be used to correlate ground and general computations of a LPDA. This relation is a special case of a more general subsumption relation between general stacks, which is not needed

---

an instance of the goal may also be read as a substitution on its variables. This allows us to consider answers of the DC program as instances of the goal. Hence our definition of an answer for the LPDA is consistent with the usual definition of answers for DC programs [Llo-87, p. 43].

here.

Definition 1 (Subsumption of ground stack by general stack)
A ground stack $\widehat{\xi} = \widehat{\mathrm{A}}_p \ldots \widehat{\mathrm{A}}_1\ \widehat{\mathrm{A}}_0$ *is an instance of the general stack* $\xi = \mathrm{A}_p.u_p \ldots \mathrm{A}_1.u_1\ \mathrm{A}_0.u_0$ *iff there is a ground substitution* σ *such that for every index* i *in* $[0..p]$ *we have the equality* $\widehat{\mathrm{A}}_i = \mathrm{A}_i u_{i+1} \ldots u_p\sigma$.
We also say that ξ subsumes $\widehat{\xi}$, *and we note* $\xi \preceq \widehat{\xi}$ *or* $\xi \preceq_\sigma \widehat{\xi}$ *or* $\widehat{\xi} = \xi\sigma$.

The following two lemmas establish a step for step correspondence between ground and general computations. Recall that a configuration is just a stack.

Lemma 2 (One-step lifting lemma for LPDA)
Let $\widehat{\xi} \vdash \widehat{\xi}'$ *be a ground computation step between two ground configurations* $\widehat{\xi}$ *and* $\widehat{\xi}'$ *of a LPDA. For any general configuration* ξ *which subsumes* $\widehat{\xi}$, *i.e.* $\xi \preceq \widehat{\xi}$, *there is a general configuration* ξ' *such that* $\xi' \preceq \widehat{\xi}'$ *and there is a general computation step* $\xi \vdash \xi'$

Lemma 3 (One-step lowering lemma for LPDA)
Let $\xi \vdash \xi'$ *be a general computation step between two general configurations* ξ *and* ξ' *of a LPDA. For any ground configuration* $\widehat{\xi}'$ *which is subsumed by* ξ', *i.e. such that* $\xi' \preceq \widehat{\xi}'$, *there is a ground configuration* $\widehat{\xi}$ *such that* $\xi \preceq \widehat{\xi}$ *and there is a ground computation step* $\widehat{\xi} \vdash \widehat{\xi}'$

Proof: The proof is straigthforward for both lemmas, proceeding by cases according to the type of transition used in the computation step. ∎

These two lemmas are extended by induction into two similar multi-steps lemmas (not given here since they are almost identical) that relate ground and general computations of arbitrary length. From these two lemmas we immediately derive the following equivalence:

Proposition 4 (Equivalence of general and ground LPDA computations)
Ground and general non-deterministic computations of a LPDA $\mathcal{A}$ *are extensionally equivalent, i.e. the ground answer set* $Answer_g(\mathcal{A})$ *is the set of all ground instances of all atoms in the set of general answer* $Answer_G(\mathcal{A})$.

3.2 Equivalence of LPDAs and DC programs

Our first purpose here is to show that LPDAs and DC programs have the same expressive power extensionally, i.e. that for every construct of one family, there is one in the other family that gives the same set of ground answers.

Our other purpose is to show that LPDAs may be used as a unifying formalism to express evaluation strategies of DC programs. In other words, we believe that DC programs are the proper declarative notation for logic programming (i.e. it expresses more simply its denotational semantics), while LPDA are a more natural way of expressing the computation strategy (i.e. the operational semantics).

3.2.1 Reduction of DC programs to LPDAs

This section is devoted to the proof of the following proposition:

Proposition 5

For every DC program $\mathcal{P}$ there is a LPDA $\mathcal{A}$ that computes the same answer extensionally.

However, as stated above *this proposition is only an existence statement* of theoretical interest, since it says nothing about the computational properties of the LPDA. The practical interest of our automata theoretic approach resides in the fact that a great variety of techniques can be developed to build and optimize such automata, very much as was done when the development of parsing technology was based on the construction of PDAs from CF grammars [AhoU-72]. The main difference in the case of DC programs is that determinism is the exception rather than the rule, and hence the driving forces in LPDA construction strategies should be quite different from those of the CF case that are aiming mainly at the preservation of determinism. Indeed it was shown by Bouckaert, Pirotte and Snelling [BouPS-75], and by later experimental work of Billot and Lang [Bil-88,BilL-88], that some constructions aiming at the preservation of determinism may have a negative effect on performances, when the resulting automaton is non-deterministic anyway.

Of course, the construction of LPDA can make use of the techniques that have been already developed for the optimization of definite clause or Datalog programs [Ull-85,BanR-86,Kow-84, BeeR-87,ZanS-86]. In fact, among other purposes, we intend LPDAs as a simple unifying formalism for expressing these constructions.

However it is not our purpose here to fully develop the LPDA construction technology, but only to set the theoretical foundation of our approach. Hence we shall only give two *very simple* LPDA construction techniques to illustrate the flexibility of the formalism and also to prove the proposition 5 above. The first is a top-down (i.e. backward chaining) LPDA, while the second is a bottom-up LPDA (i.e. forward chaining). Other LPDA constructions mimicking examples taken from the literature are presented and discussed in appendix A.

3.2.2 Construction of a backward chaining LPDA

The top-down automaton built from the DC program $\mathcal{P}$ is a LPDA that simulates OLD resolution as defined in [TamS-86]. The OLD resolution is essentially SLD resolution in which the selection function systematically selects the leftmost subgoal of the current goal (as in Prolog). For the correctness proof we assume all classical results on computation with SLD resolution, as described for example in [Llo-87]. We further assume that OLD resolution is correct — i.e. sound and complete — and that ground OLD resolution is extensionally equivalent to (general) OLD resolution. These results can be established with standard techniques of resolution based logic.

The construction of the top-down LPDA is based on the introduction of new predicate symbols $\nabla_{k,i}$, corresponding to positions between the body literals of each clause γ_k. The predicate $\nabla_{k,0}$ corresponds to the position before the leftmost literal, and so on. Literals in clause bodies are refuted from left to right. The presence of an instance of a position literal $\nabla_{k,i}(t_k)$ in the stack indicates that the first i subgoals corresponding to the body of some instance of clause γ_k have already been refuted. The argument bindings of that position literal are the partial answer substitution computed by this partial refutation.

For every clause γ_k: $A_{k,0} :- A_{k,1}, \ldots, A_{k,n_k}$, we note t_k the vector of variables occurring in the clause. Recall that $A_{k,i}$ is a literal using some of the variables in γ_k, while $\nabla_{k,i}$ is only a predicate which needs to be given the argument vector t_k to become the literal $\nabla_{k,i}(t_k)$.

Then we can define the top-down LPDA by the following transitions:

1. $\overset{\circ}{\$} \mapsto \nabla_{0,0}(t_0)\ \overset{\circ}{\$}$

2. $\nabla_{k,i}(t_k) \mapsto A_{k,i+1}\, \nabla_{k,i}(t_k)$ — *for every clause* γ_k *and for every position* i *in its body:* $0 \leq i < n_k$

3. $A_{k,0} \mapsto \nabla_{k,0}(t_k)$ — *for every clause* γ_k

4. $\nabla_{k,n_k}(t_k)\, \nabla_{k',i}(t_{k'}) \mapsto \nabla_{k',i+1}(t_{k'})$[11] — *for every pair of clauses* γ_k *and* $\gamma_{k'}$ *and for every position* i *in the body of* $\gamma_{k'}$*:* $0 \leq i < n_{k'}$

[11] If $k = k'$ then we rename the variable in $t_{k'}$ since the transition corresponds to the use of two distinct variants of the clause γ_k.

Note also that we need not define such a transition for all triples of integer k k' and i, but only for those triples such that the head of γ_k unifies with the literal $A_{k',i+1}$.

The final predicate of the LPDA is the stack predicate ∇_{0,n_0} which corresponds to the end of the body of the first "query clause" of the DC program. The rest of the LPDA is defined accordingly.

The following is an informal explanation of the above transitions:

1. *Initialization*: We require the refutation of the body of clause γ_0, i.e. of the query.
2. *Selection of the leftmost remaining subgoal*: When the first i literals of clause γ_k have been refuted, as indicated by the position literal $\nabla_{k,i}(t_k)$, then select the $i+1^{st}$ literal $A_{k,i+1}$ to be now refuted.
3. *Selection of clause* γ_k: Having to satisfy a subgoal that is an instance of $A_{k,0}$, eliminate it by resolution with the clause γ_k. The body of γ_k is now considered as a sequence of new subgoals, as indicated by the position literal $\nabla_{k,0}(t_k)$.
4. *Return to calling clause* $\gamma_{k'}$: Having successfully refuted the head of clause γ_k by refuting successively all literals in its body as indicated by position literal $\nabla_{k,n_k}(t_k)$, we return to the calling clause $\gamma_{k'}$ and "increment" its position literal from $\nabla_{k',i}(t_{k'})$ to $\nabla_{k',i+1}(t_{k'})$, since the body literal $A_{k',i+1}$ has been refuted as instance of the head of γ_k.

The computations of LPDAs thus constructed essentially mimic OLD resolution of the corresponding DC program. Hence we shall refer to it as the *OLD LPDA* associated to the DC program[12].

This LPDA is now proved correct with respect to the DC program it was constructed from. We assume that the semantics of a DC program is defined by the answers produced by *ground* OLD resolution, and we prove that *ground* computations of the LPDA give the same set of answers.

Notation: In the following lemmas and proofs, it is convenient to identify the ground argument vectors of $\nabla_{k,i}$ predicate with ground substitutions on the variables of the corresponding clause γ_k. Hence if we have the ground atom $\nabla_{k,i}(s)$, we can apply s as a substitution to any atom occurring in the clause γ_k and write for example $A_{k,j}s$.

Lemma 6 (Soundness lemma for OLD LPDA)
If there is a ground computation $\nabla_{k,i}(s)\xi \hat{\vdash}^{\star} \nabla_{k,i+1}(t)\xi$ *where* ξ *is a sequence of stack atoms never used in the computation (i.e. they is always at least one atom above it in the stack), then*

[12]The dynamic programming interpretation of this LPDA is very similar to the algorithm proposed by Tamaki and Sato in [TamS-86], when all predicates are table predicates with infinite term-depth. This is discussed in more details in section 5.

— $t{=}s$,

— *there is a ground OLD refutation of the atom* $\mathrm{A}_{k,i+1}s$.

Proof: The fact that the substitution s does not change is a trivial consequence that, *in a ground computation*, the structure of the transitions does not allow changes in the argument vector of a $\nabla_{k,i}$ predicate at a given level of the stack without popping that level.

The rest of the proof is by induction on the maximum increase of the stack depth during the computation. The first transition applied must be a push transition selecting the new ground subgoal $\mathrm{A}_{k,i+1}s$.

The base step corresponds to a maximum stack depth increase of 1, which implies no more pushing. This can only be true if the clause $\gamma_{k'}$ then selected is such that $n_{k'} = 0$. Hence the clause $\gamma_{k'}$ must be a unit clause $\mathrm{A}_{k',0}$:- of which the atom $\mathrm{A}_{k,i+1}s$ is an instance.

If the stack depth increases by more than 1, then the clause $\gamma_{k'}$ selected for the subgoal is not a unit clause. Then the computation must go through the following configurations

$$\nabla_{k',0}(s')\nabla_{k,i}(s)\xi \hat{\vdash}^{\star} \nabla_{k',1}(s')\nabla_{k,i}(s)\xi \hat{\vdash}^{\star} \cdots \hat{\vdash}^{\star} \nabla_{k',n_{k'}}(s')\nabla_{k,i}(s)\xi$$

for some ground substitution s'. By applying the induction hypothesis to each of these subcomputations, we have a refutation for each ground instance of the negative literals of the clause $\gamma_{k'}$, and hence a refutation of the ground instance of its head. ∎

Proposition 7 (Soundness of the OLD LPDA)

Any answer found by a ground computation of the OLD LPDA is an answer of the DC program from which the LPDA was constructed.

Proof: The proof is essentially the same as the induction step of the soundness lemma 6, and of course uses that lemma. Note that the final predicate of the LPDA is ∇_{0,n_0} which differs from the initial predicate of the DC program. But the answer is really the vector of arguments, to be read as an answer substitution, while the name of the predicate holding them is irrelevant. ∎

Lemma 8 (Completeness lemma for OLD LPDA)

For any ground stack configuration $\nabla_{k,i}(s)\xi$ *of the LPDA, if there is a ground OLD refutation of the literal* $\mathrm{A}_{k,i+1}s$, *then there is a ground computation* $\nabla_{k,i}(s)\xi \hat{\vdash}^{\star} \nabla_{k,i+1}(s)\xi$.

Proof: The proof goes by induction on the length of the ground refutation. It has essentially the same structure used for the soundness lemma, and it presents no difficulty since the LPDA was constructed to mimic OLD refutation. ∎

Proposition 9 (Completeness of the OLD LPDA)

Any ground answer of a DC program can be produced by a ground computation of the OLD LPDA constructed from that program.

Proof: Let s be a ground answer substitution for the DC program obtained by a ground refutation of the goal $A_{0,0}$. From the ground refutation that produces s we can extract refutations of all ground s-instances of the literals $A_{0,i}$ of the body of γ_0. Hence, using the above completeness lemma, we can exhibit a ground computation of the LPDA that produces s as result. ∎

Together, propositions 7 and 9 establish the proposition 5 stated at the beginning of this section.

3.2.3 Construction of a forward chaining LPDA

This construction of a bottom-up LPDA is given to show the flexibility of our formalism. It is a simpler variant of a *very naïve* construction published by Lang in [Lan-88b], in which we have removed an optimization aiming at reducing the polynomial complexity bounds for terminating computations.

As in the top-down case, the construction is based on the introduction of new predicate symbols $\nabla_{k,i}$, corresponding to positions between the body literals of each clause γ_k. The predicate $\nabla_{k,0}$ corresponds to the position before the leftmost literal, and so on. However, the literals of clause bodies are proved here from right to left (following the bottom-up parsing tradition, though we could proceed differently here), and thus the position literal $\nabla_{k,i}(t_k)$ in the stack indicates that the following tail of the body of clause γ_k has already been proved.

To shorten the description of the transition, we shall denote by M any atom defined by giving as argument to every predicate of the LPDA (including $\overset{\circ}{\$}$) a vector of new variables of appropriate length. It is to be used below in push transitions to indicate that the transition is independent of the top of the stack.

For every clause γ_k: $A_{k,0}$:- $A_{k,1}, \ldots, A_{k,n_k}$, we note t_k the vector of variables occurring in the clause.

The bottom-up LPDA contains the following transitions:

1. $M \mapsto \nabla_{k,n_k}(t_k)\, M$ — *for every clause* γ_k *and for every atom* M *as defined above*

2. $\nabla_{k,i}(t_k)\, A_{k,i} \mapsto \nabla_{k,i-1}(t_k)$ — *for every clause* γ_k *and*

for every position i in its body: $0 < i \leq n_k$

3. $\nabla_{k,0}(t_k) \mapsto A_{k,0}$ — *for every clause* γ_k

Here the final predicate of the LPDA is the initial predicate of the DC program, i.e. $A_{0,0}$.

In this LPDA, the presence in the stack of an atom built with a predicate of the original DC program indicates that this atom has been proved. Then the above transitions are informally explained as follows:

1. *Selection of a clause*: Ignoring the literal M on top of the stack, select an arbitrary clause γ_k whose head is to be proved; then push the position literal $\nabla_{k,n_k}(t_k)$ on the stack to indicate that none of the body literal has yet been proved (recall that they are proved from right to left).

2. *Reduction of one body literal*: The position literal $\nabla_{k,i}(t_k)$ indicates that all body literals of γ_k following the i^{th} have been proved. If we are lucky enough to have the i^{th} one $A_{k,i}$ just below the stack top (indicating it was proved earlier), then we can reduce the stack and "increment" the position literal from $\nabla_{k,i}(t_k)$ to $\nabla_{k,i-1}(t_k)$.

3. *Termination of the proof of the head of clause* γ_k: The position literal $\nabla_{k,0}(t_k)$ indicates that all literals in the body of γ_k have been proved. Hence we can replace it on the stack by the head $A_{k,0}$ of the clause, since it is now proved.

An obvious improvement to this construction would be to select only new clause instances whose head may help prove a negative literal of the current clause, but we wanted simplicity.

3.3 Reduction of LPDAs to DC programs

Proposition 10 *For every LPDA $\mathcal{A}$ there is a DC program $\mathcal{P}$ that computes the same answer extensionally.*

This reduction is mentioned for the sake of completeness. It is of theoretical interest but seems of limited importance for practical applications.

4 Dynamic Programming Execution of LPDA

Dynamic programming interpretation of a LPDA is a systematic exploration of a space of elements called *items*. This search space is a condensed representation of all possible computations of the

LPDA. It is then important to guarantee that all useful parts of that space are actually explored (cf. fairness, completeness), and that useless or redundant parts are ignored as much as possible (cf. admissibility).

This dynamic programming interpretation of a LPDA has a similar structure for both ground and general computations, because the stack substitutions of LPDA general computations become implicit and are retrieved through unification. The ground version of the algorithm can easily be derived from the general one by considering that only ground terms are computed, and by replacing unification by matching and matching by equality tests. However, for clarity, we shall present each separately, followed by its correctness proof. The presentation of these algorithms is preceded by an introduction to the main data structure used in both: the item.

We then discuss two important issues:

- *Fairness* which is the property required from the search algorithm to guarantee that no useful part of the search space is ignored, and thus ensure *completeness* of the interpretation.
- *Admissibility* which is a property required of items in the search space to give some confidence that they are unlikely to lead to redundant searches (it is usually only a necessary condition). Admissibility test are used to prune redundant parts from the search space. One practical effect is to eliminate what corresponds to some infinite branches in the standard resolution tree.

4.1 The data structure

The algorithm is based on the construction of a collection of *items*[13] that play the same role as those of Earley's algorithm, and constitute the search space. An item is a pair of atoms noted <A A'> representing a fragment of the LPDA stack[14] and a corresponding part of the computation.

[13]The word *item* is a term from [AhoU-72] and others; Earley uses the word *state* in [Ear-70].

[14]In [Die-87], Dietrich remarked that memoing lemmas in extension tables is not sufficient, and that it is necessary to memorize also configurations of the computation stack. This is precisely what the collection of items represents, with much sharing. Items play a factorization role similar to that of *solutions* in OLDT resolution [TamS-86], and of *lemmas* in SLD-AL resolution [Vie-87a]. As shown by lemmas 18 and 16, it is closely related to the notion of *context-free* subcomputation defined for the correctness proof (cf. section 4.2.2). This notion is itself close to the *subrefutations* of OLDT resolution and to the *proof segments* of SLD-AL resolution, and it is related to older notions such as *A-ancestor* in t-linear and SL resolution [KowK-71], *which aim at capturing stack-like behavior* in resolution procedures.

More precisely, in the ground case (we shall not use hatted notation here), an item $U = <A\ A'>$ may be regarded as the equivalence class of all LPDA configurations (i.e. stack contents) of the form $A\,A'\,\xi$, where ξ is any sequence of stack atoms.

It also corresponds to LPDA subcomputations starting with A' on top of the stack, and ending with A added above, without ever using A' except in the first transition that covers it with some other atom (cf. definition 11 below about "context-free subcomputations"). The proof that the dynamic programming interpretation is sound and complete relies on two lemmas (16 and 18) that formalize this intuitive interpretation of items in the ground case.

Items may also be seen as a way of decomposing the stack into elementary fragments in such a way that distinct stacks corresponding to distinct (sub)computations can share common fragments. For example, assume that the algorithm has produced among others the following collection of ground items: $<A_1\ A_2>$, $<A_2\ A_9>$, $<A_2\ A_3>$, $<A_2\ A_4>$, $<A_3\ A_4>$, $<A_3\ A_5>$, $<A_3\ A_7>$, $<A_4\ A_3>$, $<A_4\ A_5>$, $<A_5\ A_6>$ and $<A_5\ A_8>$. Then a standard non-deterministic computation of the LPDA could have produced any of the following stack configurations (among others): $A_1A_2A_9\ldots$, $A_1A_2A_3A_7\ldots$, $A_3A_5A_6\ldots$, $A_1A_2A_3A_5A_6\ldots$, $A_1A_2A_3A_5A_8\ldots$, $A_1A_2A_4A_3A_5A_8\ldots$, $A_2A_4A_3A_5A_6\ldots$, $A_1A_2A_4A_5A_8\ldots$, $A_2A_4A_3A_4A_3A_4A_3A_5A_6\ldots$, and so on. Note the circularity between A_3 and A_4 allowing arbitrary repetition of the A_3A_4 sequence in the last stack example. This is due to the two items $<A_3\ A_4>$ and $<A_4\ A_3>$. Hence the above collection of items corresponds to an infinite number of possible stack configurations.

We extend to items the definition of substitution, unification and matching, in the obvious way.

4.2 Ground dynamic programming interpretation

4.2.1 The algorithm

Initialization:

The computation is initialized by creating the *initial item* $\overset{\circ}{U} = <\overset{\circ}{\$}\ \dashv>$, where $\dashv$ indicates the bottom of the stack (it may be seen as a special proposition).

Then new items are constructed by applying the transitions of the LPDA to the already existing items as described below. Since the number of items (to be) created may be infinite, completeness requires an item examination strategy, i.e. a search strategy, that guarantees fairness, i.e. that will

never hold indefinitely a possible application of a transition. This is discussed in more details in section 4.4.

When a new item is produced, it is checked against the collection of already existing items. If it is equal to an existing item, then it is ignored. Otherwise it is added to the collection of computed items. This elimination of duplicate items is a special case of the more sophisticated admissibility tests used in the general version of the algorithm.

To each ground item $U = <\mathrm{A}\ \mathrm{A}'>$ we apply the transitions of the LPDA as follows:

Horizontal transition: $\mathrm{B} \mapsto \mathrm{C}$

This transition is applicable to the item $U = <\mathrm{A}\ \mathrm{A}'>$ iff there is a ground substitution s with domain $\mathbf{X}(\mathrm{B}, \mathrm{C})$ such that $\mathrm{B}s = \mathrm{A}$. Its application produces the new item $V = <\mathrm{C}s\ \mathrm{A}'>$.

Push transition: $\mathrm{B} \mapsto \mathrm{CB}$

This transition is applicable to the item $U = <\mathrm{A}\ \mathrm{A}'>$ iff there is a ground substitution s with domain $\mathbf{X}(\mathrm{B}, \mathrm{C})$ such that $\mathrm{B}s = \mathrm{A}$. Its application produces the new item $V = <\mathrm{C}s\ \mathrm{A}>$.

Pop transition: $\mathrm{BD} \mapsto \mathrm{C}$

This transition is applicable to the item $U = <\mathrm{A}\ \mathrm{A}'>$ iff there is a ground substitution s with domain $\mathbf{X}(\mathrm{B}, \mathrm{D}, \mathrm{C})$ such that $<\mathrm{B}\ \mathrm{D}>s = <\mathrm{A}\ \mathrm{A}'>$. Then for every item — *already existing or to be created* — $W = <\mathrm{A}'\ \mathrm{A}''>$, there is an application of the transition that produces the new item $V = <\mathrm{C}s\ \mathrm{A}''>$.

It is important to note that when several ground substitutions s may be used to apply a transition, the fair search must make sure that the transition is applied with everyone of them. Their number is in general infinite for each transition to be applied to some item(s). This problem disappears in general interpretations that use most general unifiers.

Answer:

The answer of the algorithm is the set of final atoms A (i.e. atoms with the final predicate) such that $<\mathrm{A}\ \overset{\circ}{\$}>$ is a computed item.

4.2.2 Correctness of the algorithm

The proof in the ground case requires first the definition of *context-free subcomputations* which correspond closely to the items defined in the ground dynamic programming interpretation.

Definition 11 (Context-free subcomputations)
Given two LPDA configurations ξ and ξ', we say that we have a context-free subcomputation from ξ to ξ' in n steps, and we note $\xi \sqcap\!\!\vdash^{n} \xi'$,
iff there is a sequence of $n+1$ configurations ξ_i with $0 \leq i \leq n$, such that $\xi_0 = \xi$ and $\xi_n = \xi'$,
and for any i in $[1..n]$ we have: $\xi_{i-1} \vdash \xi_i$ and $\xi_i = \xi'_i\xi$, where ξ'_i is for every i a non-empty *sequence of stack atoms.*

In other words, a context-free subcomputation is a computation that starts by pushing a new atom on top of a given stack contents ξ, and ends after n steps without ever using again the information in ξ (hence the "context-free" aspect of this subcomputation). The configurations ξ_i are said *to occur* in the subcomputation. Note that a successful computation of the LPDA is a context-free subcomputation starting with the initial stack containing only $\overset{\circ}{\$}$.

Definition 12 *We write $\xi \sqcap\!\!\vdash^{\star} \xi'$ iff there is an integer n such that $\xi \sqcap\!\!\vdash^{n} \xi'$.*

Lemma 13 *If there is a context-free subcomputation $\xi \sqcap\!\!\vdash^{\star} \xi''$ and ξ' is a configuration occurring in that computation, then we also have $\xi \sqcap\!\!\vdash^{\star} \xi'$, i.e. any initial segment of a context-free subcomputation is also a context-free subcomputation.*

Proof: Obvious from the above definitions. ∎

Note that a final segment of a context-free subcomputation may not be one, i.e. in general we do not have (in the context of the preceding lemma) $\xi' \sqcap\!\!\vdash^{\star} \xi''$.

Lemma 14 *The relation $\sqcap\!\!\vdash^{\star}$ between configurations is transitive.*

Proof: Obvious from the definition of the relation $\sqcap\!\!\vdash^{\star}$. ∎

The following lemma shows that a context-free subcomputation may be reused in distinct computations. Hence it is the basis of the dynamic programming interpretation of LPDAs.

Lemma 15 *If there is a context-free subcomputation $A\xi' \sqcap\!\!\vdash^{n} \xi A\xi'$, where A is a stack atom, ξ and ξ' are sequences of stack atoms,*
then for any sequence ξ'' of stack atoms, there is a context-free subcomputation $A\xi'' \sqcap\!\!\vdash^{n} \xi A\xi''$.

Proof: The proof is straightforward, by induction on the length of the subcomputation. ∎

Lemma 16 (Soundness lemma)

If the item $V = \langle E_1\ E_2 \rangle$ *is produced by a transition in the dynamic programming interpretation of the LPDA,*

then for some sequence ξ *of stack atoms there are two context-free subcomputations of the LPDA:* $\overset{\circ}{\$} \vdash^{\star} E_2\xi$ *and* $E_2\xi \vdash^{\star} E_1E_2\xi$.

Proof: First note that the above lemma concerns all items except the initial one $\overset{\circ}{U} = \langle \overset{\circ}{\$} \dashv \rangle$, which is not produced by a transition.

The proof goes by induction on the number of items produced before V in the dynamic programming computation, including those that are eliminated by the admissibility tests.

For the base case we have only to consider an item $\langle Cs\ \overset{\circ}{\$} \rangle$ (where s is a ground substitution) produced from the initial item with an initial push transition $\tau_0 : \overset{\circ}{\$} \mapsto C\ \overset{\circ}{\$}$. The lemma is then trivially satisfied by $\overset{\circ}{\$} \vdash^{0} \overset{\circ}{\$}$ and $\overset{\circ}{\$} \vdash^{1}_{\tau_0} Cs$.

The induction step is analyzed in three cases, according to the type of transition (horizontal, push or pop) used to produce V. The proof presents no difficulty. The pop case relies on the use of lemma 15. ∎

Proposition 17 (Soundness of ground dynamic programming interpretations)

If the final item $\langle A\ \overset{\circ}{\$} \rangle$ *is produced by the ground dynamic programming interpretation of a LPDA, then there is a ground computation of the LPDA that terminates successfully with the configuration* $A\overset{\circ}{\$}$, *i.e. it returns the answer* A.

Proof: If the item $\langle A\ \overset{\circ}{\$} \rangle$ is produced by the ground dynamic programming interpretation, then we have by the above soundness lemma $\overset{\circ}{\$} \vdash^{0} \overset{\circ}{\$}$ and $\overset{\circ}{\$} \vdash^{\star} A\ \overset{\circ}{\$}$. Hence, by definition of the relation $\vdash^{\star}$ we have $\overset{\circ}{\$} \vdash^{\star} A\ \overset{\circ}{\$}$, and since A must be a terminal atom, this is a successfully terminating computation. ∎

Lemma 18 (Completeness lemma)

Given a sequence of configurations ξ_i *for* $0 \leq i \leq g$ *and* $0 < g$, *representing the successive configurations of a ground computation of a LPDA, i.e. such that* $\xi_0 = \overset{\circ}{\$}$ *and* $\xi_{i-1} \vdash \xi_i$ *for any* i *in* $[1..g]$, *then there exists an index* g' *in* $[1..g]$, *two stack atoms* A *and* B, *and a sequence* ξ *of stack atoms such that:*

- $\xi_g = AB\xi$
- $\xi_{g'} = B\xi$
- $\xi_{g'} \vdash^{\star} \xi_g$
- *the item* $U = \langle A\ B \rangle$ *is produced by any* fair *ground dynamic programming interpretation of this LPDA.*

Proof: The proof goes by induction on the length g of the computation.

The base case, with $g = 1$, corresponds to the 1 step computation $\overset{\circ}{\$} \vdash\!\!\!\!\!\!\square_{\tau_0} Cs \overset{\circ}{\$}$, where an initial push transition $\tau_0 : \overset{\circ}{\$} \mapsto C \overset{\circ}{\$}$ is applied to the initial configuration with the ground substitution s. Then, with $g' = 0$, A = Cs, B = $\overset{\circ}{\$}$, and $\xi = \epsilon$, we satisfy the conclusions of the lemma, and in particular the construction of the item $<Cs \overset{\circ}{\$}>$ by the dynamic programming interpretation.

The induction step is proved by cases according to the type of the transition used in the last step of the computation, viz. $\xi_{g-1} \vdash_{\tau} \xi_g$. This proof presents no special difficulty. ∎

Proposition 19 (Completeness of ground dynamic programming interpretations) *If there is a ground computation of a LPDA that terminates successfully in the final configuration* $A\overset{\circ}{\$}$, *then a fair ground dynamic programming interpretation of the LPDA will produce a final item* $<A \overset{\circ}{\$}>$.

Proof: Considering the entire ground computation $\xi_0 = \overset{\circ}{\$} \vdash \xi_1 \vdash \cdots \vdash \xi_{g-1} \vdash \xi_g = A \overset{\circ}{\$}$, we can satisfy the above completeness lemma 18 only by taking (in the notation of the lemma) $g' = 0$ and B = $\overset{\circ}{\$}$ and $\xi = \epsilon$. Hence the item $<A \overset{\circ}{\$}>$ must be produced by a fair dynamic programming interpretation. Since A is a final atom, this item is also final. ∎

4.3 General dynamic programming interpretation

4.3.1 The algorithm

Initialization:

The computation is initialized by creating the *initial item* $\overset{\circ}{U} = <\overset{\circ}{\$} \dashv>$, as in the ground case.

Then new items are constructed by applying the transitions of the LPDA to the already existing items as described below. Like clauses in resolution, transitions must have their variables renamed (i.e. replaced by new unused variables) before being applied to existing items. As in the ground case, a fair search strategy is required (cf. section 4.4).

When a new item is produced, it is checked against the collection of already existing items. If it is an instance of an existing item, then it is ignored . Otherwise it is added to the collection of computed items, and all already existing items it subsumes may be removed from the collection. These tests that eliminate redundant items are called *admissibility tests* and are discussed in section 4.5.

To each item $U = <A\ A'>$ we apply the transitions of the LPDA as follows:

Horizontal transition: $B \mapsto C$

This transition is applicable to the item $U = <A\ A'>$ iff there is a substitution $s = mgu(A, B)$. Its application produces the new item $V = <Cs\ A's>$.

Push transition: $B \mapsto CB$

This transition is applicable to the item $U = <A\ A'>$ iff there is a substitution $s = mgu(A, B)$. Its application produces the new item $V = <Cs\ As>$.

Pop transition: $BD \mapsto C$

This transition is applicable to the item $U = <A\ A'>$ iff there is a unifier $s = mgu(U, <B\ D>)$. The unifier s must be chosen such that $A's$ (i.e. Ds) and Cs contain only new variables (the variables of the renamed transition $BD \mapsto C$ are considered new).

Then for every item — *already existing or to be created* — $W = <E\ E'>$ such that there is a substitution $s' = mgu(A's, E)$, there is an application of the transition that produces the new item $V = <Css'\ E's'>$.

To state things more precisely, a distinctly renamed variant of the pop transition $BD \mapsto C$ must be applied as indicated above to every pair of items U and W such that there is a unifier $s = mgu(U, <B\ D>)$ and a unifier $s' = mgu(A's, E)$. The issue is that a W item may result of a computation on another one produced by the same transisiton applied to the same item U. Hence, without careful renaming, this new W item could contain variables occurring also in $A's$. Though an apparent source of complication, this problem can be dealt with rather simply in implementations[15].

Answer:

The answer of the algorithm is the set of final atoms A (i.e. atoms with the final predicate) such that $<A\ \overset{\circ}{\$}>$ is a computed item.

---

[15] Another way to look at the application of a pop transition amounts to decomposing this application into two steps:

1. Given a pop transition $BD \mapsto C$ and an item $U = <A\ A'>$ such that there is a unifier $s = mgu(U, <B\ D>)$, the application of this pop transition to U produces a horizontal transition $Ds \mapsto Cs$.
2. Then this new horizontal transition is to be applied to all items $W = <E\ E'>$ (past or future) such that E unifies with Ds (after renaming).

This version of the algorithm does not mimic as closely the behavior of the non-deterministic LPDA, and it may thus be less intuitive. However it gives simpler formal structures, and should lead to better organized implementations.

4.3.2 Correctness of the algorithm

We now have to prove two lemmas (lifting and lowering) to extend the ground correctness results to general dynamic programming interpretations of LPDAs.

Notation: in this section we shall systematically hat ground entities to distinguish them from general entities that may contain variables.

Lemma 20 (Lifting lemma for dynamic programming interpretations)
If the item $\widehat{V}$ is a ground item produced by a ground dynamic programming interpretation of a LPDA, then any fair *general dynamic programming interpretation of the LPDA produces an item V that subsumes $\widehat{V}$, i.e. $V \preceq \widehat{V}$.*

Proof: The proof goes by induction on the number of item instances produced before $\widehat{V}$, including those discarded by the admissibility tests.

The base case is trivially satisfied when no items have been produced, that is when only the initial item is given, since it is the same initial item for all dynamic programming interpretations.

The induction steps proceeds by cases according to the nature (horizontal, push or pop) of a transition τ used in a step that produces $\widehat{V}$. We use the induction hypothesis for the item $\widehat{U}$ to which the application of this transition τ results in $\widehat{V}$. ∎

Proposition 21 (Completeness of general dynamic programming interpretations)
If an answer $\widehat{\mathrm{A}}$ is produced by a ground dynamic programming interpretation of a LPDA, then any fair general dynamic programming interpretation of that LPDA produces an answer A *that subsumes $\widehat{\mathrm{A}}$.*

Proof: Immediate from lemma 20. ∎

Lemma 22 (Lowering lemma for dynamic programming interpretations)
If the item V is produced by a general dynamic programming interpretation of a LPDA, then any ground item $\widehat{V}$ that is an instance of V (i.e. $V \preceq \widehat{V}$) is produced by any fair *ground dynamic programming interpretation of the LPDA.*

Proof: Again the proof goes by induction on the number of item instances produced before V, including those discarded by the admissibility tests.

As for the lifting lemma 20, the base case is trivially satisfied by the initial item common to all dynamic programming interpretations.

The induction step also proceeds by cases, according to the nature of a transition τ used to produce the item V. ∎

Proposition 23 (Soundness of general dynamic programming interpretations)
If an answer A *is produced by a general dynamic programming interpretation of a LPDA, then any ground instance* $\widehat{\mathrm{A}}$ *of* A *is produced by any fair ground dynamic programming interpretation of the LPDA.*

Proof: Immediate from lemma 22. ∎

4.4 Fairness

The completeness proofs rely on the fairness of dynamic programming interpretations, i.e. on the use of a *fair search strategy* that guarantees that no item remains indefinitely ignored when applicable transitions have not yet been tried, or when it still has to be used as second item for a pop transition on some other item (unless it has been deleted in the mean time by an admissibility test).

The following discussion applies to both ground and general dynamic programming interpretation. In the ground case however, it must be remembered that fairness must also encompass the choice of the ground substitution used to apply a transition, when several distinct ground substitutions are possible.

Many fair strategies may be devised. For example, we can define the *size* of an atom, or of an item by an integer valued function `size` *such that the number of elements (terms, atoms, or items) with any given size is finite.* Such a size function could be the depth of the element, or the number of function symbol occurrences it contains. Then one simple fair strategy is to always give priority to items (or pairs of items for pop transitions) that have the smallest size. This strategy is fair because the number of items that may be created with a depth less than some given constant is finite. Hence no item can be indefinitely held by items with smaller size[16].

This strategy may be understood as *breadth first on the data structures* computed by the DC program. This is much less constraining than the breadth first strategy *on the control structure*, i.e.

[16] For the same reason, our dynamic programming algorithm always terminate for Datalog programs or CF parsing. Any strategy is fair in finite interpretations.

the resolution tree, often presented as a solution to the completeness problem. With respect to control, the dynamic programming algorithm leaves open — within the above constraints — the choice between breadth-first, depth-first , or intermediate strategies as first noticed by Sheil [She-76].

Fairness may be obtained by other techniques. An obvious one is to enqueue items that are to be applied a transition, and process them with a FIFO discipline. This corresponds precisely to a breadth-first exploration of the control structure that — although exponentially more efficient — is very similar to the naïve (i.e. non dynamic-programming) breadth first interpretation of the LPDA. This is the fair strategy proposed by Porter for Earley deduction in his informal completeness proof [Por-86]. Completeness and fairness issues are not addressed at all in the original presentation of Earley deduction by Pereira and Warren [PerW-83].

The *multistage depth-first strategy* proposed by Tamaki and Sato in [TamS-86] plays a role similar to our *search strategy* for processing items. However, the *solution table* used by the algorithm is essentially a queueing structure. Hence, though it is depth-first only within one stage of computation, it tends to be rather breadth-first with respect to the total computation.

To place the discussion in a larger perspective, the fairness issue is largely independent of the use of a dynamic programming interpreter. Any fair exploration strategy on the standard resolution tree, guaranteeing that every node is eventually explored, would give answer completeness. Classical breadth-first evaluation is nothing else. The role of dynamic programming is only to avoid exploring two similar paths, path "similarity" being defined by the admissibility condition. However the "breadth-first on data-structure" strategy given above would not work on the standard resolution tree. It is fair in our context only because we make the search-space finite within the bounds of any given item size by memorizing and recognizing previously found items, thereby eliminating some looping paths.

4.5 Admissibility

The use of subsumption based admissibility tests is an established technique for pruning refutation trees. Essentially it amounts to the elimination of subgoals that are already being considered in some other part of the resolution tree, either because they are an indication of failure (infinite branches) or because the computation on the other branch may be reused and thus need not be repeated.

Admissibility tests are an essential feature of dynamic programming interpretation since the very purpose of this technique is to identify similar situations, so as to perform only once the computations required in these situations. In our context, it amounts to pruning the item space on the basis of an admissibility condition that tells whether some items are redundant.

The main difficulty is the choice of a proper admissibility condition. Simple conditions are cheap to evaluate, but are usually very restrictive and thus do not merge or eliminate computational paths efficiently. More complex condition may do a better job but are more expensive to test. The issue is to find the proper trade-off between a good pruning of the search space and the cost of the tests that perform this pruning.

With this in mind, several variants of the algorithm may be produced with distinct admissibility tests.

We call *strong admissibility* condition the requirement that a new item is not an instance of an old one. The *weak admissibility* condition is an inverse (optional) requirement leading to the elimination of old items by newly produced ones. The strong admissibility condition is close to the *local admissibility test* in SLD-AL resolution [Vie-87a] and identical to the *blocking* of an inference step in Earley deduction [PerW-83]. It is an essential feature since it can eliminate infinite loops corresponding to infinite increasing sequences of items, i.e. sequences in which each item is an instance of the previous one. Such loops typically correspond to those caused by left recursion in top-down parsing [AhoU-72].

However, this test could be replaced by variant checking, i.e. a check that the two concerned items differ only by the name of their variables. This would not lose the polynomial termination in finite interpretations (Datalog). For general DC program, the lesser pruning of the search space would leave undetected some infinite branches otherwise captured by subsumption test.

The weak admissibility condition has no counterpart in earlier proposals. However its practical usefulness is not clear: it may eliminate special case computations when a more general case is to be considered anyway, but it does not eliminate any infinite loop since there is no infinite sequence of decreasing items (the set of atoms ordered by subsumption is well founded). Thus it does not improve the termination properties. Experiments are needed to see whether, in a proper implementation, the gain due to weak admissibility elimination can exceed the cost of the corresponding tests. Weak admissibility is not distinguished from the strong one when only variant

checking is performed[17].

For ground dynamic programming interpretations, all the above admissibility tests degenerate into a simple equality check between ground items.

Our strong admissibility test corresponds to the local optimization defined in SLD-AL resolution [Vie-87b]. The global optimization of Vieille is not used here. As we see it, it eliminates subgoals that can be shown redundant with respect to a remote ancestor subgoal. In our LPDA formalism, this would amount to a test that has to explore the stack to some depth. If the depth can be arbitrary, the cost of the exploration may ruin the exponential time complexity gain obtained with the dynamic programming construction.

The issue of subsumption checking is related to the *term-depth abstraction* proposed in [TamS-86], also called *subgoal generalization* in [Vie-87a], which was not needed in our algorithm (cf. section 5.1). We see it, in our context, as an optimization device that can break, with the help of the admissibility test, some infinite computations. However when used carelessly, it may also cause other useless (possibly infinite) computations, and runs contrary to the *focus on relevant data* strategy advocated by Vieille in [Vie-87a,Vie-87c] and by most other authors in the database literature.

5 Comparison with other algorithms

5.1 Overview of the main dynamic programming algorithms

Several dynamic programming algorithm for Datalog or for Definite Clause programs have been proposed in the literature. We will restrict our detailed discussion to three algorithm [PerW-83, Por-86,TamS-86,Vie-87a,Vie-87b]. that were proposed for general Horn clauses, since the finiteness of the Datalog problem changes somewhat the importance of many issues. We do not discuss either the Extension Tables of Dietrich that did not actually lead to complete algorithms.

[17]The weak admissibility condition could cause items to be eliminated by later ones before fairness has had a chance to force applicable transitions to be actually applied. This could be repeated indefinitely with some fairness strategies, when an item can be eliminated by a later produced variant, thereby effectively defeating the chosen fairness strategy. However the well foundedness of subsumption guarantees that there is no such infinite sequence of eliminated items if an older item can be eliminated only by a strictly more general one.

Hence, if a new item is a variant of an older one, we require that the (strong) admissibility test eliminate the most recent one.

Most proposals are based on a specific evaluation strategy, usually a bottom-up evaluation optimized with a predictive component in order to reduce the size of the search space[18]. This predictive component may be added statically, as in the magic set method, by rewriting the clauses into a more sophisticated set of clauses incorporating the predictive aspect [BanMSU-85], and then evaluating this new set of clauses with the so-called semi-naïve approach (cf. section 5.3). The predictive component may also be dynamic, as in Earley deduction which dynamically mixes top-down instantiation with bottom-up reduction.

A more complex mix of top-down and bottom-up evaluation is obtained by using the *term-depth abstraction* proposed in OLDT resolution [TamS-86], also called *subgoal generalization* in SLD-AL resolution [Vie-87a,Vie-87b][19]. This mechanism controls the number of entries in the memo-table that collects (intermediate) solutions [TamS-86], or lemmas [Vie-87a,Vie-87b], corresponding to proved instances of the abstracted literals. Typically, zero term-depth produces purely bottom-up algorithms, while infinite term-depth corresponds to purely top-down resolution[20]. An increase of the term-depth corresponds to an increase of the top-down predictive component.

---

[18]Our experiments in context-free parsing [Bil-88,BilL-88], as well as earlier results by Bouckaert, Pirotte and Snelling [BouPS-75], show that predictiveness may have undesirable effects that increase the size of the search space. This is due to the greater complexity of the contextual information used in predictive computations. Hence such "optimizations" should be considered with caution, and they still require some research to be better understood.

[19]Term-depth abstraction (or subgoal generalization) amounts to a simplification of the subgoals of a refutation procedure obtained by replacing all subterms beyond a given depth by a new variable. This does not impair completeness, or termination in finite interpretations (Datalog). However, as noted at the end of section 4.5, such generalizations may cause useless computations, and could even cause an infinite loop in an otherwise terminating computation. Potential loss of efficiency due to the use of generalization is mentioned by Vieille in [Vie-87b].

The *restriction* used by Shieber in [Shi-85] is similar to subgoal generalization. His version of Earley deduction with restriction is itself very similar to OLDT resolution. The use of restriction avoids infinite loops in the predictive (i.e. top-down) part of his algorithm, which essentially computes the equivalent of OLDT entries (and their number is finite for finite term-depth). The bottom-up part of the algorithm always terminates because a non-infinitely ambiguous context-free backbone is assumed in the natural language grammars/programs for which the algorithm is intended.

[20]Actually Tamaki and Sato do not allow infinite term depth: they need a bounded term-depth to have a bounded number of entries in their solution table; this in turn guarantees the fairness of their multi-stage depth-first strategy, since the solution table is used as a queueing structure, with one queue per entry. However, for their extension to negation in stratified databases [SekI-88], Seki and Itoh use our variant with infinite term-depth, or more precisely without term-depth at all, because their restriction to finite interpretations eliminates problems of termination and hence of fairness.

While Tamaki and Sato limit their OLDT algorithm to the OLD resolution strategy, Vieille adds another control mechanism by allowing an arbitrary (homogeneous) subgoal selection function, provided it is *local* (see below). Furthermore, Vieille allows variations on the search strategy. Thus by varying the various parameters of his algorithm, Vieille claims that he can obtain (approximately) several well-known procedures for recursive databases [Vie-87b]. Similar variations seem also possible for Earley deduction, though they are not considered in [PerW-83].

This very rapid survey calls for several remarks concerning these algorithms:

- They have limited possibilities in the choice of the execution strategy, except possibly for Vieille SLD-AL resolution.
- The execution strategy of the algorithms is hidden, hard to analyze, and often tightly mixed with the features that realize the dynamic programming aspects. This is typically the case with the term depth abstraction in OLDT resolution, which plays a role both in the execution strategy and in the fairness strategy.
- Except for Earley-deduction, the data structures are rather complex and difficult to analyze.
- Furthermore (again excluding Earley deduction) these structures exhibit asymmetries that are not present in the original problem. Some nodes in the partial resolution trees directly dominate the subtree that refutes their goal, while others are just references to the lemma/solution table. This is furthermore dependent on the order in which the clauses are processed. We believe that this situation betrays the fact that the fundamental structures have not been exhibited. It certainly makes the implementation more complex.

Our techniques avoid these problems by completely separating the free choice of the execution strategy, which is embodied in the construction of the LPDA, from the implementation of the dynamic programming procedure.

- The fine grained LPDA instructions (i.e. transitions) allow a very free specification of the execution strategy. We illustrate this in appendix A by giving two LPDAs whose dynamic programming interpretation mimics Earley deduction and OLDT resolution with zero term-depth.
- The simplicity of the instruction set of the LPDA leads to a very simple dynamic programming implementation, without any interaction with the chosen execution strategy. The data

structures (items) are simple and present no unjustified biases. This is conducive to tractable and efficient implementation as was sketched in a previous paper [Lan-88b].

It must be noticed that the selection function used in OLD resolution, and hence in OLDT resolution, is a *local selection function* in the sense of Vieille [Vie-87a,Vie-87b], i.e. a function that always selects subgoals in the remaining part of the body of the clause instance most recently introduced in the resolution computation. This is essentially a LIFO behavior (last introduced, first processed) on the processing of subgoals. Vieille's construction is also restricted to resolution with local selection function. Though the notion of selection function is not evident in Earley deduction as it is presented in [PerW-83], its behavior is rather similar to OLDT resolution with infinite term-depth (cf. section 3.2.2 and appendix A), and it can be described directly as push-down based (cf. section 5.2). This convergence and our own experience of the problem lead to the conjecture that *these dynamic programming constructions are possible because there are corresponding non-deterministic algorithms using their memory with a LIFO discipline.*

Even if that conjecture were not verified, the above remarks show that the LPDA formalism is probably an appropriate vehicle for unifying previously published constructions. Furthermore the language Prolog itself uses a local selection function — it is actually OLD resolution — and its implementation model is a non-deterministic stack machine. Hence the same decomposition between LPDA construction and interpretation could be used for Prolog technology interpreters. Or conversely, techniques developed for optimizing Prolog program could be adapted to the production of LPDA for dynamic programming interpretation. Finally, it should be possible to combine cleanly Prolog type backtrack interpretation with dynamic programming interpretation, so as to trade memory requirements for execution speed. This is similar to the proposal of Tamaki and Sato and Vieille to have only some predicate as "table predicate", and corresponds to the classical use of *memo-functions* in functional programming [Bir-80].

5.2 Comparison with Earley deduction

Earley deduction is the simplest of the earlier proposals, and shares several properties with our algorithm, such as simplicity and regularity of its structure. Arguably, it could well be used as the basic interpretation machine for DC program. Then one would essentially follow the same pattern as we proposed:

- first compile the given DC program into another program better optimized for interpretation,

- then apply Earley deduction to the optimized version.

The first step of this procedure was actually proposed to implement many optimization techniques in recursive databases [BanMSU-85,ZanS-86]. As is usual in databases, the final fix-point computation technique is a bottom-up evaluation of the transformed program[21].

While this proposal may be workable, there is no evidence that all optimization techniques may be easily or cleanly expressed by transformation of DC program into a variant program intended for interpretation by Earley deduction. To our knowledge, no results along those lines has been published, nor has Earley deduction been formally analyzed with respect to correctness, complexity or implementation techniques. Furthermore, as noted earlier (cf. footnote in section 5.1), there is no evidence that the predictive approach of Earley deduction is optimal. Hence it is certainly necessary to compare it to other approaches within a uniform framework such as the one we propose.

Another drawback of Earley deduction is that we do not know of any other simple computation engine (like our LPDA) which would be its non-deterministic counterpart and could serve as simpler reference and as intermediate step for analysis and proofs. It seems indeed that the natural non-deterministic engine corresponding to Earley deduction is a push-down machine on which full clauses are pushed before being reduced (popped) subgoal by subgoal, the head being popped together with the subgoal below it, that caused the rule to be pushed in the first place. But this is a push-down engine.

Thus we believe that the main justification for our model is that push-down mechanisms are hidden in many of the structures proposed in the deduction literature, and particularly that concerning dynamic programming evaluators. Expliciting these aspects should improve our understanding of the phenomena. It should also help bridge the gap with functional programming where the push-down model is standard, especially in the first order case to which we limited our investigations.

[21] This indicates that these transformations could be considered for our compilation into LPDA, which is to some extent a bottom-up machine (cf. section 5.3). However, they often make the hypothesis that the final interpreter works in ground mode. They can probably be much improved and simplified in general interpreters.

5.3 Comparison with Bottom-up resolution

Bottom-up resolution implemented with a semi-naïve[22] saturation technique has been proposed by several authors(mostly in the database literature [BanMSU-85,RoeLK-86,ZanS-86]) as the basic mechanism for evaluating recursive queries or DC programs, after rewriting them in an optimized form that uses top-down information. This approach originally intended for Datalog has been extended to full DC programs by some authors [Ram-88,KifL-88].

We show here that the set of transitions of a LPDA is very similar to a set of clauses intended for bottom-up evaluation. However the LPDA transitions have a little more expressive power (in push transitions) which allows to express simply a tighter control of the computation.

We shall assume that the DC program is put in a normal form such that a clause body has at most two literals. This restriction is natural if we want to have a simple machine, and this simple form can be easily obtained without distortion of the original DC grammar. Then the three types of clauses may be read as transitions of a LPDA as follows:

- the unit clause C :- reads as a collection of push transitions: M $\mapsto$ CM where M is any fully uninstantiated atom (as many as there are predicates),
- the clause C :- B reads as the horizontal transition B $\mapsto$ C
- the clause C :- BD reads as the pop transition BD $\mapsto$ C

The execution of this LPDA precisely corresponds to bottom-up resolution (cf. section 3.2.3).

Ground dynamic programming interpretation of this LPDA performs essentially the same computation as the semi-naïve bottom-up evaluation of the original program. However, LPDA obtained in this way do not make use of the contextual facility of push transitions that allows their application only when an atom matching their left-hand side has been found: the M atoms are dummies that play no active role in the push transitions of this LPDA.

Considering things the other way around, it is a non-trivial task to translate an arbitrary LPDA into a collection of clauses whose bottom-up evaluation mimics the steps of the LPDA computation. It requires a predictive data-flow analysis of the LPDA computation, which is very similar to the "Magic Templates" construction proposed by Ramakrishnan [Ram-88]. We believe this to be a

[22] In database terminology [BanR-86], "semi-naïve evaluation" refers to a ground bottom-up incremental saturation without any optimization. The incremental (or differential) technique, that avoids recomputing twice the same tuple in the same way, is a dynamic programming procedure.

correct interpretation of Ramakrishnan's statement that his algorithm can execute a DC program "bottom-up, while computing no facts that are not also generated by the Prolog execution".

6 Conclusion

The LPDA engine is not the most general formalism for describing the operational semantics of DC programs. There are computations that may be expressed as resolution rules and do not follow our model. However it seems to be at this time the simplest and most used formalism for the interpretation of DC programs, though this has remained implicit in the literature, both for Prolog implementations and for dynamic programming algorithms based on resolution with a local selection function. This important role is not surprising: a non-deterministic push-down automaton is a natural device for exploring And-Or trees with the push-down to memorize And-nodes and the non-determinism to handle Or-nodes.

We have shown here that when it is made explicit, the PDA model clarifies and unifies a number of constructions and makes them theoretically and practically more tractable. It emphasizes the role of locality of interpretation with respect to the data structure, which is essential for the algorithms to have a good time complexity.

In addition to its conceptual simplicity, the modularity of the approach emphasizes the possible variations in the operational strategies, and isolates independent choices in independent parts of the compilation/interpretation process. As was the case for general context-free parsing, we expect it to be the unifying framework in which to express existing dynamic programming algorithms and to study new optimization techniques.

Indeed, the constructions presented here have already been applied to produce straightforwardly a time $\mathcal{O}(n^6)$ Earley-type parser for *Tree Adjoining Grammars* [Lan-88c]. At the time of this writing, the only published Earley-like TAG parser claims $\mathcal{O}(n^9)$ time complexity, and the problem was considered difficult by its authors Shabes and Joshi [SchJ-88][23].

This formalization is quite open to extensions. An obvious one is the combination of dynamic

[23]Other dynamic programming parsers for TAGs have been published, with $\mathcal{O}(n^6)$ [VijJ-85] and $\mathcal{O}(n^4)$ [Har-88] time complexities. However they are both CYK-like algorithms [Hay-62,Kas-65,You-66] that always attain their worst complexity, unlike Earley-like algorithms. Furthermore, the latter algorithm requires the transformation of the TAG grammar into a normal form.

programming and backtrack interpretation[24]. Seen from a different angle, our results could be used for the implementation of memo functions in functional languages with call by unification [Mau-88]. Another clear extension is the generalization to the treatment of negation in stratified programs, as was done for SLD-AL [KemT-88] and OLDT [SekI-88] resolution.

Finally, it may be noticed that the techniques developed here are independent of the interpretation domain. Though we have used the classical Herbrand interpretations for our logic programs, we could have used other interpretations domains. Examples are found in the data types proposed for Prolog extensions, such as the LOGIN formalism of Aït-Kaci and Nasr [AïtN-86], or as the Prolog-II of Colmerauer [Col-82]. Other examples are the feature structures used in some linguistic formalisms [Shi-84,RouK-86], or the non-standard interpretations used for static analysis of Prolog programs [AbrH-87]. Another possibility is to consider higher-order terms (lambda terms) as proposed by Miller and Nadathur [MilN-87], though the undecidability of unification may then raise additional problems.

Acknowledgements: I wish to thank Laurent Vieille for initiating me to recursive queries, for supplying many useful references, and for his comments on earlier drafts. I also gratefully acknowledge the constructive comments of Catriel Beeri, Véronique Donzeau-Gouge, Francis Dupont, Laurent Hascoet, Gérard Huet, Neil Jones, Michel Mauny.

The example in appendix B was produced with a prototype implementation due to Eric Villemonte de la Clergerie and Alain Zanchetta [VilZ-88].

Extended abstracts of this work, dated respectively February 1988 and May 1988, were submitted to the Fifth International Conference Symposium on Logic Programming (LP'88) held in Seattle (Washington) in August 1988, and to the International Conference on Fifth Generation Computer Systems (FGCS'88) to be held in Tokyo (Japan) in November 1988.

This work was supported in part by the Eureka Software Factory project.

[24]This problem is similar to the handling of local determinism in [Lan-74].

References

[AasPS-88] Aasa, A.; Petersson, K.; and Synek, D. 1988 Concrete Syntax for Data Objects in Functional Languages. *Proc. of the 1988 ACM Conf. on Lisp and Functional Programming*, Snowbird (Utah), pp. 96-105.

[AbrH-87] Abramsky, S.; and Hankin, C. (Eds.) 1987 *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited (Pub.).

[AhoHU-68] Aho, A.V.; Hopcroft, J.E.; and Ullman, J.D. 1968 Time and Tape Complexity of Pushdown Automaton Languages. *Information and Control* 13: 186-206.

[AhoU-72] Aho, A.V.; and Ullman, J.D. 1972 *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, Englewood Cliffs, New Jersey.

[AïtN-86] Aït-Kaci, H.; and Nasr, R. 1986 Login: A logic Programming Language with Built-in Inheritance. *Journal of Logic Programming* 3: 185-215.

[BanMSU-85] Bancilhon, F.; Maier, D.; Sagiv, Y.; and Ullman, J.D. 1985 *Magic Sets: Algorithms and Examples.* MCC Technical Report DB-168-85.

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

[BeeR-87] 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).

[Bil-88] Billot, S. 1988 *Analyseurs Syntaxiques et Non-Déterminisme.* Thèse de Doctorat, Université d'Orléans la Source, Orléans (France).

[BilL-88] Billot, S.; and Lang, B. 1988 *The structure of Shared Forests in Ambiguous Parsing.* In preparation.

[Bir-80] Bird, R.S. 1980 Tabulation Techniques for Recursive Programs. *Computing Survey* 12(4): 403-417.

[BouPS-75] Bouckaert, M.; Pirotte, A.; and Snelling, M. 1975 Efficient Parsing Algorithms for General Context-Free Grammars. *Information Sciences* 8(1): 1-26

[Coh-88] Cohen, J. 1988 A View of the Origins and Development of Prolog. *Communications of the ACM* 31(1) :26-36.

[Col-78] Colmerauer, A. 1978 Metamorphosis Grammars. in *Natural Language Communication with Computers*, L. Bolc ed., Springer LNCS 63. First appeared as *Les Grammaires de Métamorphose*, Groupe d'Intelligence Artificielle, Université de Marseille II, 1975.

[Col-82] Colmerauer, A. 1982 *PROLOG-II — Manuel de Référence et Modèle Théorique.* Groupe d'Intelligence Artificielle, Université de Marseille - Luminy.

[DeR-71] DeRemer, F.L. 1971 Simple LR(k) Grammars. *Communications ACM* 14(7): 453-460.

[Des-88] Despeyroux, T. 1988 *Typol: a Formalism to Implement Natural Semantics.* INRIA Tech. Report 94.

[Die-87] Dietrich, S.W. 1987 Extension Tables: Memo Relations in Logic Programming. *Proc. of 1987 Symposium on Logic Programming*: 264-272, San Francisco (California).

[Ear-70] Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 13(2): 94-102.

[GriP-65] Griffiths, I.; and Petrick, S. 1965 On the Relative Efficiencies of Context-Free Grammar Recognizers. *Communications ACM* 8(5): 289-300.

[Har-88] Harbusch, K. 1988 *Effiziente Analyse Natürlicher Sprache mit TAGs.* Bericht Nr. 38, FB 10 - Informatik IV, Universität des Saarlandes, Saarbrücken (F.R.G.).

[Hay-62] Hays, D.G. 1962 Automatic Language-Data Processing. In *Computer Applications in the Behavioral Sciences*, (H. Borko ed.), Prentice-Hall, pp. 394-423.

[HeeKR-88] Heering, J.; Klint, P.; and Rekers, J. 1988 Incremental Generation of Parsers. *ESPRIT Project 348 - GIPE, Third Annual Review Report*, Annex D12.A5.

[Imm-82] Immerman, N. 1982 Relational Queries Computable in Polynomial Time. *Proc. of the* 14^th *ACM/SIGACT Symposium* :147-152.

[Kah-87] Kahn, G. 1987 Natural Semantics. *Proc. of Symp. on Theoretical Aspects of Computer Science*, Passau (Germany). Also INRIA Research Report 601.

[KapB-82] Kaplan, R.M.; and Bresnan J. 1982 Lexical-Functional Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*, ch. 4, pp. 173-281, J. Bresnan (ed.), The MIT Press.

[Kas-65] 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.

[Kay-80] Kay, M. 1980 Algorithm Schemata and Data Structures in Syntactic Processing. *Proceedings of the Nobel Symposium on Text Processing*, Gothenburg.
Also: Tech. Report CSL-80-12, Xerox Palo-Alto Research Center.

[Kay-84] Kay, M. 1984 Unification in Grammar. *Proc. of the First Internat. Workshop on Natural Language Understanding and Logic Programming*: 233-240, Rennes (France), V. Dahl and P. Saint-Dizier (eds.), Elsevier Science Pub. (North-Holland), 1985.

[KemT-88] Kemp, D.B.; and Topor R.W. 1988 Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases. *Proc of LP'88, Fifth Internat. Conference Symposium on Logic Programming*, Seattle (Washington).

[KifL-88] Kifer, M.; and Lozinskii, E.L. 1988 SYGRAF: Implementing Logic Programs in a Database Style. *IEEE Trans. on Software Engineering* 14(7):922-935.

[Knu-65] Knuth, D.E. 1965 On the Translation of Languages from Left to Right. *Information and Control*, 8: 607-639.

[KowK-71] Kowalski, R.; and Kuehner, D. 1971 Linear Resolution with Selection Function. *Artificial Intelligence* 2: 227-260.

[Kow-84] Kowalski, R. 1984 *Logic for Problem Solving.* North-Holland (New York).

[Lan-71] Lang, B. 1971 Parallel Non-deterministic Bottom-up Parsing (abstract). Proc. of International Symposium on Extensible Languages, Grenoble (France), *SIGPLAN Notices* 6(12): 56-57.

[Lan-74] 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).

[Lan-88a] Lang, B. 1988 Parsing Incomplete Sentences. *Proc. of the* 12^{th} *Internat. Conf. on Computational Linguistics (COLING'88)* Vol. 1 :365-371, D. Vargha (ed.), Budapest (Hungary).

[Lan-88b] Lang, B. 1988 Datalog Automata. *Proc. of the* 3^{rd} *Internat. Conf. on Data and Knowledge Bases*, C. Beeri, J.W. Schmidt, U. Dayal (eds.), Morgan Kaufmann Pub., pp. 389-404, Jerusalem (Israel).

[Lan-88c] Lang, B. 1988 *The Systematic Construction of Earley Parsers: Application to the Production of an* $\mathcal{O}(n^6)$ *Earley Parser for Tree Adjoining Grammars.* In preparation.

[Llo-87] Lloyd, J.W. 1987 *Foundations of Logic Programming.* (second edition), Springer-Verlag.

[MaiW-88] Maier D.; and Warren, D.S. 1988 *Computing with Logic — Logic Programming with Prolog.* The Benjamin/Cummings Publishing Company.

[Mak-87] Makowsky, J.A. 1987 Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples. *Journal of Computer and System Sciences* 34:266-292.

[Mau-88] Mauny M. 1988 Personnal Communication.

[MilN-87] Miller, D.; and Nadathur, G. 1987 A Logic Programming Approach to Manipulating Formulas and Programs. *Proc. of the 1987 Symp. on Logic Programming*:379-388, San-Francisco (California).

[PerW-80] Pereira, F.C.N.; and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231-278.

[PerW-83] Pereira, F.C.N.; and Warren, D.H.D. 1983 Parsing as Deduction. *Proceedings of the* $21^{s}t$ *Annual Meeting of the Association for Computational Linguistics*: 137-144, Cambridge (Massachusetts).

[Phi-86] 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.

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

[Pra-75] Pratt, V.R. 1975 LINGOL — A Progress Report. In *Proceedings of the* 4^{th} *IJCAI*: 422-428.

[Ram-88] Ramakrishnan, R. 1988 Magic Templates: A Spellbinding Approach to Logic Programming. *Proc. of the Fifth Internat. Conf. Symp. on Logic Programming — LP'88*, Seattle (Washington).

[Rob-65] Robinson, J.A. 1965 A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23-41.

[RoeLK-86] Roehmer, J.; Lescoeur, R.; and Kerisit, J.M. 1986 The Alexander Method: A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing* 4(3) :273-285.

[RouK-86] Rounds, W.C.; and Kasper, R. 1986 A Complete Logical Calculus for Record Structures Representing Linguistic Information. *Proc. of IEEE Symp. on Logic in Computer Science* :38-43, Cambridge (Massachusetts).

[SekI-88] Seki, H.; and Itoh, H. 1988 An Evaluation Method of Stratified Programs under the Extended Closed World Assumption. *Proc of LP'88, Fifth Internat. Conference Symposium on Logic Programming*, Seattle (Washington).

[SchJ-88] Schabes, Y.; and Joshi, A.K. 1988 Yves Aravind An Earley-type Parsing Algorithm for Tree Adjoining Grammars. *Proc. of the* 26^{th} *Annual Meeting of the Association for Computational Linguistics*: 258-269, Buffalo (New York).
Also: Technical Report MS-CIS-88-36, LINC LAB 113, Dept. of Computer and Information Science, University of Pennsylvania, 1988.

[She-76] 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).

[Shi-84] Shieber, S.M. 1984 The Design of a Computer Language for Linguistic Information. *Proc. of the* 10^{th} *Internat. Conf. on Computational Linguistics — COLING'84*: 362-366, Stanford (California).

[Shi-85] Shieber, S.M. 1985 Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. *Proceedings of the* $23^{r}d$ *Annual Meeting of the Association for Computational Linguistics*: 145-152, Chicago (Illinois).

[TamS-86] Tamaki, H.; and Sato, T. 1986 OLD Resolution with Tabulation. *Proc. of* $3^{r}d$ *Internat. Conf. on Logic Programming*, London (UK), Springer LNCS 225: 84-98.

[Tom-87] Tomita, M. 1987 An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[Ull-85] Ullman, J.D., 1985 Implementation of Logical Query Languages for Databases. *ACM transactions on Database Systems*, 10(3):289-321.

[VanK-76] 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.

[Vie-87a] Vieille, L. 1987 Database-Complete Proof Procedures Based on SLD Resolution. *Proc. of the* 4^{th} *Internat. Conf. on Logic Programming*: 74-103, Melbourne (Australia).

[Vie-87b] Vieille, L. 1987 *Recursive Query Processing: The power of Logic.* Tech. Report TR-KB-17, European ComputerIndustry Research Center (ECRC), Munich (West Germany).

[Vie-87c] 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.

[VijJ-85] Vijay-Shankar, K.; and Joshi, A.K. 1985 Some Computational Properties of Tree Adjoining Grammars. *Proceedings of the* $23^{r}d$ *Annual Meeting of the Association for Computational Linguistics*: 145-152, Chicago (Illinois).

[VilZ-88] Villemonte de la Clergerie, E.; and Zanchetta, A. 1988 *Evaluateur de Clauses de Horn.* Rapport de Stage d'Option, Ecole Polytechnique, Palaiseau (France).

[Voi-88] Voisin, F. 1988 Une Version Ascendante de l'Algorithme d'Earley. *International Workshop on Programming Languages Implementation and Logic Programming*, Orleans (France).

[You-66] Younger, D.H. 1967 Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2): 189-208

[ZanS-86] Zaniolo, C.; and Saccà, D. 1986 Rule Rewriting Methods for Efficient Implementations of Horn Logic. Foundations of Logic and Functional Programming, Workshop Proc., Trento (Italy), M. Boscarol, L. Carlucci Aiello, G. Levi (eds.), Springer LNCS 306: 114-139.

A Other examples of LPDA construction techniques

As stated in section 5, the LPDA constructions given in that section were meant mainly as an existence proof, and were chosen primarily for their simplicity.

To illustrate the generality and the flexibility of the LPDA formalism, we give here two other examples mimicking respectively Earley deduction and the OLDT algorithm for zero term-depth. Note that this mimicking is limited to performing similar computation steps, but possibly with different constraints on their order. This is to be expected since, as intended, the description of the LPDA gives no hints about the search strategy or even the admissibility criterion used.

The following constructions remain rather simple examples. More complex ones, with sophisticated optimizations, would be beyond the scope of this paper.

A.1 OLDT resolution with zero term-depth

The OLDT resolution of Tamaki and Sato [TamS-86] is parameterized by a term-depth limit on so-called table predicates. The construction given below mimics this algorithm in the case where all predicates are table predicates with term-depth zero.

This is essentially a bottom-up algorithm, with a light top-down component that limits activation to "useful" clauses (see below).

The $n + 1$ clauses of the DC program are defined as γ_k : $A_{k,0}$:- $A_{k,1}, \ldots, A_{k,n_k}$ with γ_0 defining the goal. For each clause γ_k we note t_k the vector of variables occurring in the clause.

For each predicate P of the DC program, we define 3 stack predicates P′, P″ and P‴, and we note x_P a vector of new variables with a length equal to the arity of P.

For each atom $A_{k,i}$ occurring in a clause γ_k, the notations $A'_{k,i}$, $A''_{k,i}$ and $A'''_{k,i}$ denote the same atom where the predicate symbol, say P, has been replaced respectively by P′, P″ and P‴.

We define in the LPDA the transitions:

1. $\overset{\circ}{\$} \mapsto A'_{0,0} \overset{\circ}{\$}$
2. $P'(x_P) \mapsto \nabla_{k,0}(t_k)\, P'(x_P)$ — *for every predicate* P *and for every clause* γ_k *defining* P.
3. $\nabla_{k,i}(t_k) \mapsto A'_{k,i+1}\, \nabla_{k,i}(t_k)$ — *for every clause* γ_k *and for every position* i *in its body:* $0 \leq i < n_k$

4. $\nabla_{k,n_k}(t_k) \mapsto A''_{k,0}$ — *for every clause* γ_k

5. $P''(x_P)\,P'(x_P) \mapsto P'''(x_P)$ — *for every predicate* P

6. $A'''_{k,i+1}\,\nabla_{k,i}(t_k) \mapsto \nabla_{k,i+1}(t_k)$ — *for every clause* γ_k *and for every position* i *in its body:* $0 \leq i < n_k$

The final predicate of the LPDA is the stack predicate $A'''_{0,0}$.

The following is an informal explanation of the transitions defined above:

1. State the query in the stack.
2. For a P (sub)query, choose a defining clause γ_k that could give an answer,
3. and subquery successively for each atom in the clause body. The position predicate $\nabla_{k,i}$ indicates that the subqueries for the first i atoms have been simultaneously satisfied.
4. When a common answer substitution θ has been found for all atoms in the body of γ_k, then the head instance $A_{k,0}\theta$ is proved. It is represented here by $A''_{k,0}$.
5. Given a solution to the most general query on P — i.e. $P(x_P)$ — represented here by a proved P'' atom, and given a P query represented here by a P' atom, if they have a common instance represented here by the P''' atom, then this instance is an answer to the query P'.
6. Having found for the subgoal $A_{k,i+1}$ an answer represented by the atom $A'''_{k,i+1}$, we proceed to the next subgoal.

The algorithm is essentially bottom-up in principle. It looks top-down because the original algorithm is so organized that it considers only the predicates that can be "activated" in the body of a clause defining another activatable predicate. Note that *no* binding information is being passed as argument in such activations.

This is almost like considering all predicates in the call graph from the initial query. However we do not consider calls (subgoals) following an initial body segment that cannot be consistently answered, thus making the whole clause irrelevant anyway (since there are no initial bindings). This last point disallows static determination of the useful predicates and clauses, thus forcing the top-down structure in our imitation of the original algorithm.

A.2 Earley deduction

The LPDA mimicking Earley deduction is very similar to the OLD LPDA defined in section 3.2.2. The main difference is that, while OLD resolution removes a subgoal that is being refuted, Earley deduction keeps it until the refutation is successful.

That is, in Earley deduction, a subgoal is used to activate clauses defining its predicate in the same sense as in the zero term-depth OLD LPDA above, but the bindings of the activating subgoals are used to instantiate the clause. Heads of instantiated clauses are taken as subqueries, and after successful processing they are kept as unit clauses for reduction (resolution) of subgoals from other clause instances.

The $n + 1$ clauses of the DC program are defined as γ_k : $A_{k,0} :- A_{k,1}, \ldots, A_{k,n_k}$ with γ_0 defining the goal. For each clause γ_k we note t_k the vector of variables occurring in the clause.

For each predicate P of the DC program, we define 2 stack predicates P′ and P″.

For each atom $A_{k,i}$ occurring in a clause γ_k, the notations $A'_{k,i}$ and $A''_{k,i}$ denote the same atom where the predicate symbol, say P, has been replaced respectively by P′ and P″.

We define in the LPDA the transitions:

1. $\overset{\circ}{\$} \mapsto A'_{0,0} \overset{\circ}{\$}$
2. $A'_{k,0} \mapsto \nabla_{k,0}(t_k)\, A'_{k,0}$ — *for every clause* γ_k
3. $\nabla_{k,i}(t_k) \mapsto A'_{k,i+1}\, \nabla_{k,i}(t_k)$ — *for every clause* γ_k *and for every position* i *in its body:* $0 \leq i < n_k$
4. $\nabla_{k,n_k}(t_k)\, A'_{k,0} \mapsto A''_{k,0}$ — *for every clause* γ_k
5. $A''_{k,i+1}\, \nabla_{k,i}(t_k) \mapsto \nabla_{k,i+1}(t_k)$ — *for every clause* γ_k *and for every position* i *in its body:* $0 \leq i < n_k$

The final predicate of the LPDA is the stack predicate $A''_{0,0}$.

The following is an informal explanation of the transitions defined above:

1. State the query $A_{0,0}$ in the stack, represented by $A'_{0,0}$.
2. Choose clause γ_k to prove the goal sitting on top of the stack: the head $A_{k,0}$ of the clause γ_k must subsume it.

3. Then subquery successively for each atom in the body of γ_k. The position predicate $\nabla_{k,i}$ indicates that the subqueries for the first i atoms have been consistently proved.

4. When a common answer substitution θ has been found for all atoms in the body of γ_k, then the head instance $A_{k,0}\theta$ is proved. It is represented here by $A''_{k,0}$.

5. Having proved an instance of the i-th atom in the body of γ_k consistently with the proofs of previous ones, move to the next body atom to be subqueried by transition 3, unless there is none left in which case an answer to a previous subquery is produced by transition 4.

Though it essentially captures the working of Earley deduction, the above LPDA differs in several details.

First the original Earley deduction allows an *arbitrary* selection function for every new choice of a subgoal, which we have not allowed (but could allow) in our construction above. However it is usually inefficient to allow too much non-determinism in the selection function: this is because the efficiency of the dynamic programming is due to a factorization of identical subcomputations, and this factorization is possible only if all steps of these subcomputations are in the same order. The authors probably meant that the selection function is a parameter of the algorithm. Note that the OLD/Prolog selection function used in the above construction was also used in the examples of the original paper [PerW-83].

Another point is that the "inference blocking" admissibility test of the original algorithm implicitly merges equivalent instantiated subclauses[25], even when they come from different original program clauses. This is not done above, since the $\nabla_{k,i}$ predicates standing for subclauses are distinct for distinct clauses. Achieving this result here would require a much more careful construction of the transitions, though the issue is partially dealt with by the variation on the interpretation of pop transitions described in the footnote of section 4.3.1.

Note finally that our transitions are statically produced, while the subclauses are produced dynamically in Earley deduction. A consequence is that we have to produce statically many transitions (very many if the LPDA is to be optimized), while some of these transitions may never be actually needed in computations. It is not clear to us how much a static analysis of the DC program might improve this situation. However, compiling the clauses into transitions on a "by need" basis would not be difficult. It is in fact an interesting development of the techniques

[25] We call *instantiated subclause* a program clause where some already proved body atoms have been removed, while the clause has been instantiated with the substitution produced by these proofs.

presented here, which, in accordance with the modularity principles advocated in this paper, should be analyzed independently. Similar work was done for an Earley-like algorithm constructed by dynamic programming interpretation of an LR(1) parser where the transition are incrementally computed when needed [HeeKR-88].

The same remarks apply to the above OLDT LPDA, and also more generally to DC program optimization by static clause rewriting, as in magic-set like techniques.

B An Implemented Example

The following example has been produced with a prototype implementation realized by Eric Villemonte de la Clergerie and Alain Zanchetta [VilZ-88]. The computer printout (in teletype characters) has been edited for inclusion in this appendix.

The definite clause program to be executed is:

```
Clauses:  q(f(f(a))):-.
          q(X1):-q(f(X1)).
Query:    q(X2)
```

Note that a search for all solutions in a backtrack evaluator would not terminate.

The solutions found by the computer are: `X2 = f(f(a))`

`X2 = f(a)`

`X2 = a`

These solutions were obtained by first compiling the DC program into an LPDA, and then interpreting this LPDA with the general dynamic programming algorithm defined in section 4.3.1.

The definite clause program is repeated in figure 3. The LPDA transitions produced by its compilation according to the OLD LPDA construction of section 3.2.2 are given in figure 2. The collection of items used by the computation is given in the figure 4.

In the transitions printout of figure 2, each predicate name `nabla.i.j` stands for our $\nabla_{i,j}$.

According to the OLD LPDA construction of section 3.2.2, the final predicate should be `nabla.0.1`. For better readability we have added a horizontal transition to a final predicate noted `answer`.

```
********* PUSH Transitions B->BC ***********

   predicate :nabla.2.0
nabla.2.0(X1) -> q(f(X1)) nabla.2.0(X1)

   predicate :nabla.0.0
nabla.0.0(X2) -> q(X2) nabla.0.0(X2)

   predicate :dollar0
dollar0() -> nabla.0.0(X2) dollar0()

********* Horizontal Transitions B->C ******

   predicate :q
q(f(f(a))) -> nabla.1.0()
q(X1) -> nabla.2.0(X1)

   predicate :query
query(X2) -> nabla.0.0(X2)

   predicate :nabla.0.1
nabla.0.1(X2) -> answer(X2)

********* POP Transitions BD->C ************

   predicate :nabla.2.1
nabla.2.1(X1) nabla.0.0(X2) -> nabla.0.1(X2)
nabla.2.1(X4) nabla.2.0(X1) -> nabla.2.1(X1)

   predicate :nabla.1.0
nabla.1.0() nabla.0.0(X2) -> nabla.0.1(X2)
nabla.1.0() nabla.2.0(X1) -> nabla.2.1(X1)

   predicate :nabla.0.1
nabla.0.1(X3) nabla.0.0(X2) -> nabla.0.1(X2)
nabla.0.1(X2) nabla.2.0(X1) -> nabla.2.1(X1)
```

Figure 2: Transitions of the LPDA.

```
Clauses:  q(f(f(a))):-.
          q(X1):-q(f(X1)).
Query:    q(X2)
```

Figure 3: The Definite Clause program.

```
dollar0() , ()()
nabla.0.0(X5) , dollar0()
q(X6) , nabla.0.0(X6)
nabla.2.0(X7) , nabla.0.0(X7)
nabla.1.0() , nabla.0.0(f(f(a)))
q(f(X8)) , nabla.2.0(X8)
nabla.0.1(f(f(a))) , dollar0()
nabla.2.0(f(X9)) , nabla.2.0(X9)
nabla.1.0() , nabla.2.0(f(a))
nabla.2.1(f(a)) , nabla.0.0(f(a))
nabla.0.1(f(a)) , dollar0()
q(f(f(X10))) , nabla.2.0(f(X10)) *
nabla.2.1(f(a)) , nabla.2.0(a)
nabla.2.1(a) , nabla.0.0(a)
nabla.0.1(a) , dollar0()
answer(a) , dollar0()
answer(f(a)) , dollar0()
answer(f(f(a))) , dollar0()

* subsumed by: q(f(X8)),nabla.2.0(X8)
```

Figure 4: Items produced by the dynamic programming interpretation.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique