Chart Parsers as Inference Systems for Fixed-Mode Logic Programs*

David A. Rosenblueth

IIMAS, UNAM
Apdo. 20-726, 01000 Mexico D.F.
drosenbl@servidor.unam.mx

Abstract

Logic programs resemble context-free grammars. Moreover, Prolog's proof procedure can be viewed as a generalization of a simple top-down parser with backtracking. This simple parser has disadvantages that motivated the design of more sophisticated parsing methods. As similar disadvantages occur in Prolog's proof procedure, it may be desirable to develop other proof procedures for logic programs than the one used by Prolog. The resemblance between definite clauses and productions suggests looking at parsing to develop such procedures. We obtain proof procedures for fixed-mode logic programs, based on "chart" parsers.

Our approach concentrates on transforming (fixed-mode) logic programs rather than the parser. We first add unification to a chart parser obtaining a proof procedure for programs severely restricted in their syntax, in which the body of the clauses denotes the composition of binary relations: "chain" programs. We then show how to transform fixed-mode programs into chain form. We arrive at proof procedures that avoid some nonterminating loops as well as the recomputation of some partial results.

1 Introduction

A logic program consists of clauses that look like the productions of a context-free grammar. This suggests connections between proof procedures and parsers. In fact, Prolog's proof procedure can be regarded as a generalization of a simple parser with backtracking. Although this language has seen numerous applications, its execution mechanism

*A previous version of this paper was presented at the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, 1992.

has several disadvantages. For instance, if such a mechanism finds an infinite branch of the derivation tree, it enters a nonterminating loop. In addition, every time a partial result is needed, it is recomputed. Thus, it may be desirable to develop new proof procedures for logic programs.

Simple parsers with backtracking also enter nonterminating loops easily and recompute partial results. This has motivated the design of other more sophisticated parsing methods, such as Earley's algorithm [8].

Pereira and Warren have obtained [18] a proof procedure for logic programs, which is inspired by Earley's parser, but the result is inefficient compared with Prolog. It uses subsumption, which is NP-complete [10]. We concentrate on transforming the program as opposed to the parser, and obtain proof procedures for fixed-mode logic programs, based on chart parsing. Our proof procedures do not use subsumption and may be more efficient than Pereira and Warren's.

Chain programs. By using difference lists to represent strings, a logic program can be restricted to coincide with the productions of a context-free grammar. Hence, for this class of logic programs, parsers *are* proof procedures. Such a class, however, only has the expressive power of context-free grammars. Assuming that we are interested in having a programming language, this suggests generalizing such programs without losing the close similarity with grammars. We will do so by allowing the body of clauses to denote the composition of arbitrary binary relations; such programs will be said to have "chain" form. It is then easy to obtain a proof procedure for chain programs: we add term "matching" (one way unification) to a chart parser. Prolog programs, however, are not normally written in chain form. Thus, we consider programs in a larger class (fixed-mode programs) and transform [20] them into chain form.

Fixed-mode programs. A "mode" for a subgoal is the subset of arguments that are variables at the time the subgoal is selected. Thus, the mode depends on the derivation tree for a program and a goal. When we refer to a "fixed-mode logic program," we actually mean a program and a goal such that with Prolog's computation rule all subgoals with the same predicate symbol have the same mode. This class of programs is important because it includes many programs occurring in practice. Prolog programmers, for instance, often write fixed-mode programs. There are two reasons for this. On the one hand, it is often more natural to think procedurally first [15]. On the other hand, there are obstacles to the invertibility of logic programs, such as the use of built-in predicates and the lack of the occurs check.

Overview of the paper. The rest of this paper is organized as follows. Section 2 reviews chart parsers. Section 3 converts such parsers into inference systems for chain programs by adding term matching. Section 4 deals with a method for transforming fixed-mode logic programs into chain form, thus making these inference systems applicable to fixed-mode logic programs. Section 5 elaborates the inference systems of Section 3 and increases their efficiency for programs resulting from our transformation into chain form. Section 6 compares our work with several existing methods. Section 7 concludes this paper summarizing our results. We include in Appendix D the Prolog code of a proof procedure resulting from having added a search strategy to one of our inference systems.

2 Chart parsers

Chart parsers [11] are methods for parsing strings of context-free languages, which can be viewed as a generalization of Earley's algorithm [8]. These parsers normally assume, with no loss of generality, that the grammar has been rewritten so that it only contains productions of the form:

$$A \to c$$
$$A \to B_1 B_2 \dots B_n$$

Here, and throughout the paper, we adhere to the convention of using lower-case letters for terminals, and upper-case letters for nonterminals. In addition, we denote by R the set of productions of the grammar and by Σ the alphabet of terminals.

The main idea in chart parsing is the concept of an "edge." We can view this concept as a four-place predicate:

$$\mathit{edge}(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1}, \dots, B_n]) \qquad\qquad i > 0,\ n \geq 0$$

where $\mathbf{x}$ and $\mathbf{y}$ denote the strings x and y, respectively, and $A, B_i, B_{i+1}, \dots, B_n$ are the nonterminals that generate the languages $A', B'_i, B'_{i+1}, \dots, B'_n$, respectively; that is, $A' = \{x \in \Sigma^* : A \stackrel{*}{\Rightarrow} x\}$ and $B'_j = \{x \in \Sigma^* : B_j \stackrel{*}{\Rightarrow} x\}$. This predicate is intended to hold when y is a suffix of x, and:

$$A' \supseteq \{x - y\} \cdot B'_i \cdot B'_{i+1} \cdot \dots \cdot B'_n$$

where $x - y$ represents, as in difference lists, the string resulting from removing y from x, and "$\cdot$" denotes language product.[1]

Note that a production $A \to B_1 B_2 \dots B_n$ means that $A' \supseteq B'_1 \cdot B'_2 \cdot \dots \cdot B'_n$. Thus, an edge can be thought of as a partially applied production.

A set of edges usually called *chart*, is often represented by a directed graph. In this graph, the nodes are normally labeled with the symbols of the string being parsed. For us, however, it will be more convenient to label the nodes with lists denoting *strings*. We represent each edge:

$$\mathit{edge}(\mathbf{x}, \mathbf{y}, A, [B_1, B_2, \dots, B_n])$$

[1] The product of languages L and M is defined as: $L \cdot M = \{xy : x \in L \,\&\, y \in M\}$.

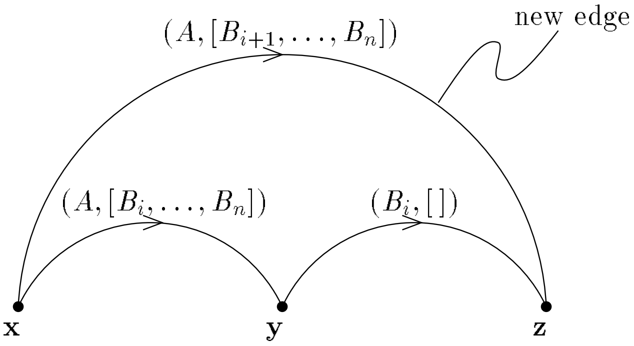

Figure 1: *The fundamental rule.*

by an arc from the node labeled with $\mathbf{x}$ to the node labeled with $\mathbf{y}$. This arc is labeled with the pair $(A, [B_1, B_2, \ldots, B_n])$.

There are rules deriving new edges from the productions of the grammar and from the edges derived previously. Let us denote the input string by $\mathbf{x}_0$, the empty string by $[\,]$, and the initial symbol by S. Then, the edge $edge(\mathbf{x}_0, [\,], S, [\,])$ asserts that the input string belongs to the language generated by the grammar. We shall concentrate on top-down chart parsing, but refer the reader to [11, 13] for more comprehensive explanations of chart parsing.

The fundamental rule. The operation sometimes called *fundamental rule* takes two edges and produces a third edge. From $edge(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1} \ldots, B_n])$ and $edge(\mathbf{y}, \mathbf{z}, B_i, [\,])$, this rule produces $edge(\mathbf{x}, \mathbf{z}, A, [B_{i+1}, \ldots, B_n])$, as illustrated in Figure 1. Intuitively, this rule can be viewed as the application of one symbol to the partially applied production represented by the first edge.

$$\frac{edge(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1}, \ldots, B_n]) \qquad edge(\mathbf{y}, \mathbf{z}, B_i, [\,])}{edge(\mathbf{x}, \mathbf{z}, A, [B_{i+1}, \ldots, B_n])}$$

The soundness of this rule follows from the definition of language product.

The top-down rule. The fundamental rule produces a new edge from existing edges. Thus, we need a rule that takes into account the productions of the grammar. The *top-down rule* generates, from an edge of the form $edge(\mathbf{x}, \mathbf{y}, A, [B_i, \ldots, B_n])$, and a production $B_i \to C_1 \ldots C_k$, an edge of the form $edge(\mathbf{y}, \mathbf{y}, B_i, [C_1, \ldots, C_k])$:

$$\frac{edge(\mathbf{x}, \mathbf{y}, A, [B_i, \ldots, B_n]) \qquad (B_i \to C_1 \ldots C_k) \in R}{edge(\mathbf{y}, \mathbf{y}, B_i, [C_1, \ldots, C_k])}$$

Note that the edges added by the top-down rule are vacuously true. Intuitively, the reason for adding these edges, is to allow for the possible future application of B_i to the edges of the form $edge(\mathbf{x}, \mathbf{y}, A, [B_i, \ldots, B_n])$. This rule is depicted in Figure 2 (left).

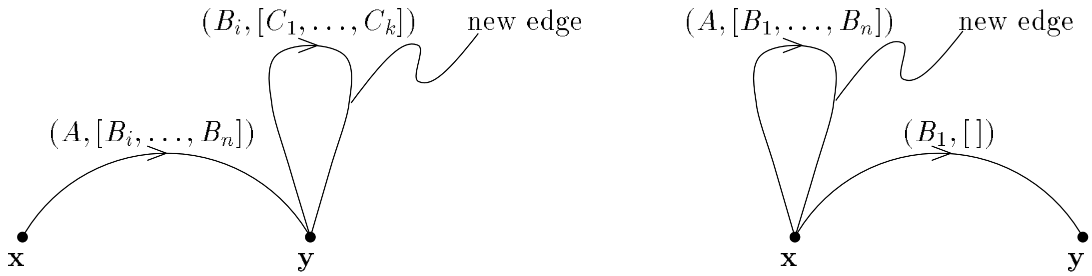

Figure 2: *The top-down rule (left) and the bottom-up rule (right).*

Given a parse tree with a leaf B_i, this rule starts the expansion of B_i by creating an edge labeled with a pair having B_i on its left-hand side. Hence a parse tree is built by expanding the leaves with nonterminals, which is a construction of parse trees from the root towards the leaves.

The initialization rules. There are usually rules for creating the first edges of a chart. For the top-down parser, one such rule is:

$$\frac{(S \to A_1 A_2 \ldots A_n) \in R}{\mathit{edge}(\mathbf{x}_0, \mathbf{x}_0, S, [A_1, A_2, \ldots, A_n])}$$

In addition, we must take care of productions of the form $S \to c$.

$$\frac{(S \to c) \in R}{\mathit{edge}(\mathbf{x}_0, [\,], S, [\,])}$$

where $\mathbf{x}_0$ denotes c.

The base-extension rule. Finally, observe that, except for productions of the form $S \to c$, we have no rule considering the productions having a terminal on their right-hand side. Such productions are normally used to generate a set of edges often called the *base* of the chart, consisting of one edge labeled with $(A, [\,])$ going from a list denoting the string cz to a list denoting a string z, where $c \in \Sigma$ and $z \in \Sigma^*$, for each production $A \to c$. Some chart parsers create the base in its totality at initialization, and require the whole string before parsing starts. It is also possible, however, to create the base incrementally as the chart is built. For us, a parser that builds the base incrementally is preferable. The reason is that in our inference systems, sequences of terms play the role of strings, and we do not know *a priori* any sequence of terms that may be the frontier of a proof tree. A parser which builds the base incrementally will allow us to use the part of the chart already constructed as a guide for extending the base. A common base-extension rule for such chart parsers [13] is:

$$\frac{edge(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1}, \ldots, B_n]) \qquad (B_i \to c) \in R}{edge(\mathbf{y}, \mathbf{z}, B_i, [\,])}$$

where $\mathbf{y}$ and $\mathbf{z}$ denote, respectively, the strings cz and z. This rule varies in the literature, and we think that for the purposes of this paper, any of these variations could have been chosen, since similar inference systems are obtained.

Agenda. Note that a rule may produce more than one edge, so that nondeterministic choices may appear. Normally, chart parsers contain, in addition to the chart being built, an auxiliary data structure, called the *agenda*, storing all the edges to which the rules have not yet been applied. Hence, when the rules generate edges, these are added to the agenda, from which they are removed one by one. The discipline with which the edges are added to and removed from the agenda determines a search strategy in the space produced by the nondeterministic choices. For instance, if the agenda is represented by a stack, then depth-first traversal is obtained; if represented by a queue, breadth-first traversal is obtained instead.

Observe also that whenever an edge is removed from the agenda, such an edge may already exist in the chart. Thus, a test for membership is needed to prevent the parser from entering into a nonterminating loop.

The bottom-up rule. Parse trees are built from the leaves towards the root when the so-called *bottom-up rule* is used instead of the top-down rule.

$$\frac{edge(\mathbf{x}, \mathbf{y}, B_1, [\,]) \qquad (A \to B_1 \ldots B_n) \in R}{edge(\mathbf{x}, \mathbf{x}, A, [B_1, \ldots, B_n])}$$

Figure 2 (right) shows the bottom-up rule.

Given a parse tree T with root B_1, this rule creates an edge labeled with a pair $(A, [B_1, \ldots, B_n])$, thus adding an elementary subtree with root A on top of T. This is a construction from the leaves towards the root.

As with the base-extension rule, we have arbitrarily chosen this particular version [11, page 197] of the bottom-up rule.

Example. Figure 3 exhibits a chart created by a parser using the top-down rule for the grammar with productions:

$$\begin{aligned}
&\hat{a} \to k_0\ \hat{a}\ k_1 \\
&\hat{a} \to \langle [1], [2] \rangle \\
&k_0 \to \langle [\,], [1,2] \rangle \\
&k_1 \to \langle [1], [\,], [2] \rangle
\end{aligned}$$

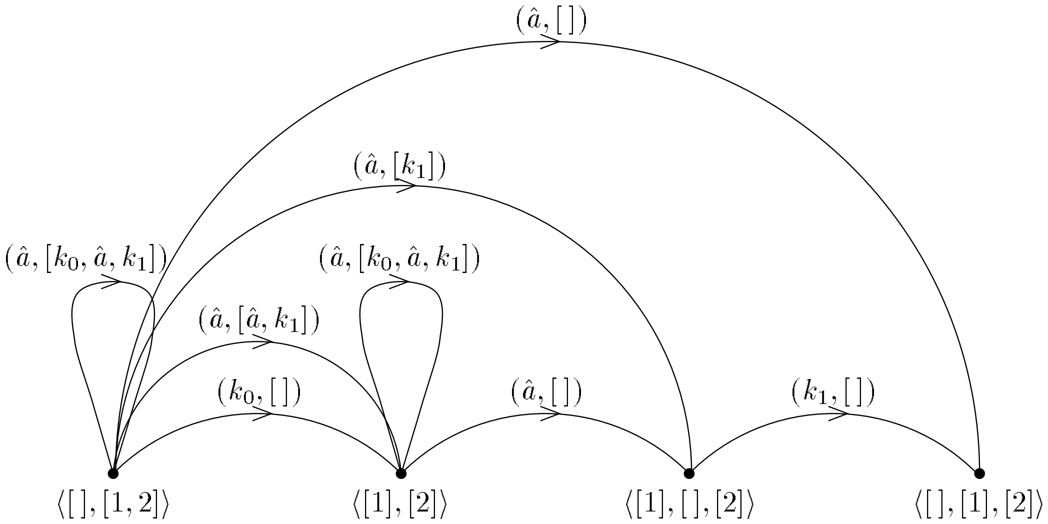

Figure 3: *A chart constructed with the top-down rule.*

and the input string $\langle [\,], [1, 2] \rangle$ $\langle [1], [2] \rangle$ $\langle [1], [\,], [2] \rangle$ $\langle [\,], [1], [2] \rangle$. Terminals have been enclosed in angled brackets. The last symbol $\langle [\,], [1], [2] \rangle$ is not part of the string itself, but rather an end marker. We have labeled the nodes with terminals, instead of strings, so that we can use this example later to illustrate a chart created by an inference system when decomposing the list $[1, 2]$.

3 An inference system for chain programs

In this section, we will slightly modify the top-down chart parser, to obtain an inference system for a class of programs generalizing the difference-list representation of context-free grammars, which we call "chain" programs. This modification consists in adding term matching to the base-extension rule.

3.1 Inference rules

Difference lists. Using difference lists, a logic program P_G can be associated with a context-free grammar G as follows. For each production of the form:

$$A \rightarrow B_1 B_2 \ldots B_n$$

P_G has a clause of the form:

$$a(X_0, X_n) \leftarrow b_1(X_0, X_1),\ b_2(X_1, X_2),\ \ldots,\ b_n(X_{n-1}, X_n) \tag{1}$$

For each production of the form:

$$A \rightarrow c$$

P_G has a clause of the form:

$$a([c|X], X) \leftarrow \tag{2}$$

P_G has the property that $P_G \models s(\mathbf{x}, [\,])$ if and only if $x \in L(G)$, where $\mathbf{x}$ and $[\,]$ denote x and the empty string, respectively. Hence, the chart parser we described in the previous section can be viewed as an inference system for such a logic program.

Charts and chain programs. With a programming language having only these clauses we cannot compute all computable functions. But if we generalize (2) to:

$$a(t, t') \leftarrow \tag{3}$$

where t and t' are terms such that $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, we can. (Here, and throughout the sequel, we use $\mathrm{var}(t)$ to denote the set of variables occurring in term t.) This can be shown, for instance, by associating a flowchart with a logic program in such a way that both have the same set of computations [6].

We define a *chain program* as a logic program having only clauses of the form (1) and (3). A *chain goal* is a goal of the form $\leftarrow p(\mathbf{x}, Z)$, where $\mathbf{x}$ is a ground term and Z a variable.

We will now obtain an inference system for a chain program and a chain goal. To this purpose, we associate edges with clauses as follows. The edge:

$$\mathit{edge}(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1}, \ldots, B_n])$$

represents the clause:

$$a(\mathbf{x}, X_n) \leftarrow b_i(\mathbf{y}, X_i),\ b_{i+1}(X_i, X_{i+1}),\ \ldots,\ b_n(X_{n-1}, X_n)$$

With this representation, the fundamental rule becomes a resolution step, and the initialization and the top-down rules become instantiations of program clauses.

The base-extension rule. A natural generalization of the base-extension rule is:

$$\frac{\mathit{edge}(\mathbf{x}, \mathbf{y}, A, [B_i, B_{i+1}, \ldots, B_n]) \qquad b_i(t, t') \leftarrow \text{ is a variant of a clause in } P}{\mathit{edge}(\mathbf{y}, t'\theta, B_i, [\,])}$$

where θ is the unifier for $\mathbf{y}$ and t. This is a resolution step. By the syntactic condition on the occurrence of variables in unit clauses of chain programs, $t'\theta$ is ground because $\mathbf{y}$ also is.

Charts and nondeterministic computations. Note that unlike context-free grammars, chain programs can cause the base-extension rule to be nondeterministic, in the sense that there may be more than one way to extend the computation.

Example. A portion of the chart created by an inference system using the top-down rule for the program:

$$\hat{a}(U_0, U_3) \leftarrow k_0(U_0, U_1),\ \hat{a}(U_1, U_2),\ k_1(U_2, U_3) \tag{4}$$

$$\hat{a}(\langle St, Z\rangle, \langle St, [\,], Z\rangle) \leftarrow \tag{5}$$

$$k_0(\langle St, [W|Z]\rangle, \langle [W|St], Z\rangle) \leftarrow \tag{6}$$

$$k_1(\langle [W|St], X, Y\rangle, \langle St, [W|X], Y\rangle) \leftarrow \tag{7}$$

and the goal $\leftarrow \hat{a}(\langle [\,], [1,2]\rangle, Z)$ was shown in Figure 3. For simplicity, we have not drawn the complete chart, which can be thought of as having several planes, where the other answers for the program and the goal would appear.

3.2 Soundness and completeness of td-inference

We now state the soundness and completeness for the top-down chart parser converted into an inference system. The soundness and completeness of the corresponding inference system based on a bottom-up chart parser can be established in a similar way.

Let P be a chain program, C a chart, and E an edge. The chart $C \cup \{E\}$ is *td-derived* from P and C if E is inferred using any of the top-down, the fundamental, or the base-extension rules. A *td-derivation* for P and a chart C_0 is a (finite or infinite) sequence $C_0, C_1, \ldots$ of charts such that C_{i+1} is td-derived from C_i. The variants of the unit clauses in the base-extension rule do not have variables in common with any clause appearing in the td-derivation up to C_i. A *td-derivation* for $P \cup \{G\}$, where G is a chain goal, is a td-derivation for P and the chart obtained by applying the initialization rule using P.

By observing that all inference rules of td-inference are sound, we can show the soundness of this inference system.

Theorem 3.1 (Soundness of td-inference) *Let P be a chain program and G a chain goal. If there exists a td-derivation for $P \cup \{G\}$ ending in a chart containing* $\mathrm{edge}(\mathbf{x}, \mathbf{y}, p, [\,])$, *then $P \models p(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ and $\mathbf{y}$ are ground terms.*

To show the completeness, we first need an auxiliary lemma, of which we give a sketch proof in Appendix B. (A more detailed proof of the completeness of td-inference is in [1].)

Lemma 3.2 *Let P be a chain program. If $P \models q(\mathbf{x}, \mathbf{y})$, then there exists a td-derivation for P and* $\{\mathrm{edge}(\mathbf{w}, \mathbf{x}, p, [q, \ldots])\}$ *ending in a chart containing* $\mathrm{edge}(\mathbf{x}, \mathbf{y}, q, [\,])$, *where $\mathbf{w}$, $\mathbf{x}$, and $\mathbf{y}$ are ground terms.*

Using the initialization rules and Lemma 3.2, we can prove the completeness of td-inference.

Theorem 3.3 (Completeness of td-inference) *Let P be a chain program. If $P \models s(\mathbf{x}, \mathbf{y})$, then there exists a td-derivation for $P \cup \{\leftarrow s(\mathbf{x}, Z)\}$ ending in a chart containing* $\mathrm{edge}(\mathbf{x}, \mathbf{y}, s, [\,])$, *where $\mathbf{x}$ and $\mathbf{y}$ are ground terms, and Z is a variable.*

4 Conversion of fixed-mode programs into chain form

We have transformed chart parsers into inference systems for chain programs. However, logic programs are not normally written in chain form. In this section, we observe that it is possible to convert a fixed-mode logic program into chain form. The resulting program is logically implied by an extension of the original program.

First, we define the class of programs that can be transformed by our method. Then we prove, for a particular example, the correctness of the resulting program. A sketch of a proof for the general case is in Appendix A; a more detailed proof can be found in [20].

4.1 Directed programs

Directed form. In a fixed-mode program, the arguments of each predicate are partitioned into those used as "input" and those used as "output" of an operation. Thus we assume, with no loss of generality, that in each predicate, all input arguments have been grouped into one argument, and all output arguments into another one. We write the input argument first, and the output argument second.

The mode for a subgoal actually depends on the derivation tree for the program and the goal. Because we wish to describe our transformation syntactically, we will approximate the property of a program and a goal having fixed modes with syntactic conditions as follows.

A definite clause of the form:

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n) \qquad n \geq 0$$

where:

1. $\mathrm{var}(t_i) \cap \mathrm{var}(t_j) = \emptyset$, for $i, j = 0, \ldots, n$ and $i \neq j$;
2. $\mathrm{var}(t'_i) \subseteq \mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_i)$, for $i = 0, \ldots, n$;
3. for $i = 0, \ldots, n$, each variable occurring in t'_i occurs only once in t'_i

is a *directed clause.* A *directed program* is a logic program having only directed clauses. Condition 1 causes the term constructed when a subgoal succeeds to have an effect only on the input of other subgoals. Condition 2 causes the input argument of all selected subgoals to be ground if the input of the initial goal is also ground and subgoals are selected in a left-to-right order. We include Condition 3 only for technical reasons. This is a minor restriction that considerably simplifies both stating our transformation and proving it correct. We call these "directed" programs because we can visualize the binding of a variable as flowing from one occurrence to subsequent occurrences.

We shall need various axioms. As with program clauses, we assume that each axiom is implicitly universally quantified with respect to its variables.

Normally, an SLD derivation is either successful, failed, or infinite. Sometimes, however, we shall use derivations that end in a clause that could possibly be resolved with a program clause. We shall refer to these derivations as *partial derivations.*

A partial derivation with a single-subgoal initial goal yields a *conditional answer* [26]. Such an answer is a clause in which the head is the subgoal in the initial goal of that derivation with the composition of the substitutions applied to it, and the body is the set of subgoals in the last goal of that derivation.

4.2 Example

We illustrate our method with the following program for decomposing a list. It defines the usual *append* relation, but its arguments have been grouped in such a way that its input is a list composed of the two lists in its output. $a(\langle Z\rangle, \langle X, Y\rangle)$ holds if Z is the list that can be decomposed into the list X followed by the list Y. The angled brackets $\langle\ \rangle$ are an alternative notation for ordinary brackets [ ], that we use to group input and output arguments. We do this for clarity.

$$a(\langle Z\rangle, \langle [\,], Z\rangle) \leftarrow \tag{8}$$

$$a(\underbrace{\langle [W|Z]\rangle}_{t_0}, \underbrace{\langle [W|X], Y\rangle}_{t_1'}) \leftarrow a(\underbrace{\langle Z\rangle}_{t_0'}, \underbrace{\langle X, Y\rangle}_{t_1}) \tag{9}$$

We shall convert (9) into chain form. This process can be motivated as follows.

In a clause of the form:

$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1),\ p_2(X_1, X_2),\ \ldots,\ p_n(X_{n-1}, X_n)$$

the $p(U, V)$ predicates hold over *complete* descriptions of states (see [6]). The $a(U, V)$ predicate, on the other hand, does not hold over such a complete description. To see this, consider clause (9). To be able to construct the list $[W|X]$ (in t_1') from the body of this clause, it is necessary to remember the term with which the W unifies in the list $[W|Z]$ (in t_0). This lack of information in the arguments of the body of (9) prevents us from representing a computation by the composition of the relation denoted by $a(U, V)$ with itself. Hence, we must provide the missing information to the arguments. A common technique in the implementation of programming languages with destructive assignment for recording values needed in subsequent steps of a computation is the use of a stack. This suggests the possibility of storing the term unifying with W in a list that is treated as a stack. We thus define the predicate:

$$\hat{a}(\langle St_0|U\rangle, \langle St_1|V\rangle) \leftrightarrow St_0 = St_1 \ \&\ a(U, V) \tag{10}$$

Although both St_0 and St_1 represent the same stack, it will be convenient to keep two different names for this term, so that the input of this new predicate shares no variables with the output. Later we will see why we wish clauses in which the input and the output of their atoms share no variables.

We will also use the *standard equality theory.* This theory consists of the following axioms:

$$
\begin{aligned}
&X = X \leftarrow \\
&X = Y \leftarrow Y = X \\
&X = Z \leftarrow X = Y,\ Y = Z \\
&f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \leftarrow X_1 = Y_1, \ldots, X_n = Y_n \\
&p(U, V) \leftarrow U = X,\ V = Y,\ p(X, Y)
\end{aligned}
$$

which are called, reflexivity, symmetry, transitivity, function substitutivity, and predicate substitutivity. Note that the last two are actually *axiom schemas*, since an axiom is included for every function and predicate symbol, respectively.

First, we derive a clause in which the input and the output of the atoms have no variables in common:

$$a(\langle [W|Z] \rangle, \langle [W'|X], Y \rangle) \leftarrow W = W',\ a(\langle Z \rangle, \langle X, Y \rangle) \tag{11}$$

This clause can be obtained as a conditional answer, starting from the goal $\leftarrow a(U, V)$ and using function substitutivity to disassemble the term $\langle [W|X], Y \rangle$, and reflexivity to assemble it with W' instead of W.

Next, we can proceed as follows. Unfolding[2] (11) on the "if" part of the definition of $\hat{a}$ (10) we obtain:

$$\hat{a}(\langle St_0, [W_0|Z_0] \rangle, \langle St_1, [W_1|X_1], Y_1 \rangle) \leftarrow St_0 = St_1,\ W_0 = W_1,\ a(\langle Z_0 \rangle, \langle X_1, Y_1 \rangle)$$

Subsequently, we fold the "iff" version $[U|V] = [U'|V'] \leftrightarrow U = U' \ \&\ V = V'$ of the function substitutivity axiom for the list-constructor function symbol:

$$\hat{a}(\langle St_0, [W_0|Z_0] \rangle, \langle St_1, [W_1|X_1], Y_1 \rangle) \leftarrow [W_0|St_0] = [W_1|St_1],\ a(\langle Z_0 \rangle, \langle X_1, Y_1 \rangle)$$

and fold the definition of $\hat{a}$:

$$\hat{a}(\langle St_0, [W_0|Z_0] \rangle, \langle St_1, [W_1|X_1], Y_1 \rangle) \leftarrow \hat{a}(\langle [W_0|St_0], Z_0 \rangle, \langle [W_1|St_1], X_1, Y_1 \rangle) \tag{12}$$

Now the head (W_0) of the first list in the original clause can be thought of as being removed from that list, and pushed onto the stack, then being removed from the stack

[2]In program-transformation terminology, the "unfold" operation is a resolution step. The "fold" operation replaces the subgoals that unify with a conjunction of atoms by a single atom using a definition.

with another name (W_1) and finally added to the front of the result of decomposing the tail of the original list.

The fact that in (12) the inputs share no variables with the outputs allows us to fold the following definitions of k_0 and k_1:

$$k_0(U,V) \leftrightarrow \exists St_0 \exists W_0 \exists Z_0.[\langle St_0, [W_0|Z_0]\rangle = U \ \& \ \langle [W_0|St_0], Z_0\rangle = V]$$
$$k_1(U,V) \leftrightarrow \exists St_1 \exists W_1 \exists X_1 \exists Y_1.[\langle [W_1|St_1], X_1, Y_1\rangle = U \ \& \ \langle St_1, [W_1|X_1], Y_1\rangle = V]$$

in the following clause:

$$\begin{aligned}\hat{a}(U_0,U_3) \leftarrow\ & \langle St_0, [W_0|Z_0]\rangle = U_0,\\ & \langle [W_0|St_0], Z_0\rangle = U_1,\\ & \langle [W_1|St_1], X_1, Y_1\rangle = U_2,\\ & \langle St_1, [W_1|X_1], Y_1\rangle = U_3,\\ & \hat{a}(U_1,U_2)\end{aligned}$$

which is a logical consequence of (12) and the standard equality theory. The resulting clause is:

$$\hat{a}(U_0,U_3) \leftarrow k_0(U_0,U_1),\ \hat{a}(U_1,U_2),\ k_1(U_2,U_3)$$

Using a result found, for instance, in [24, pages 57, 58] we can prove that the fold steps preserve all models of the program.

It may not be practical to transform a program with fold and unfold operations. The chain form of a directed program may be obtained in a more straightforward manner based on the theorem in Appendix A.

The chain form of the *append* program is:

$$\begin{aligned}&\hat{a}(U_0,U_3) \leftarrow k_0(U_0,U_1),\ \hat{a}(U_1,U_2),\ k_1(U_2,U_3)\\ &\hat{a}(U_0,U_1) \leftarrow k_0'(U_0,U_1)\\ &k_0(\langle St, [W|Z]\rangle, \langle [W|St], Z\rangle) \leftarrow\\ &k_1(\langle [W|St], X, Y\rangle, \langle St, [W|X], Y\rangle) \leftarrow\\ &k_0'(\langle St, Z\rangle, \langle St, [\,], Z\rangle) \leftarrow\end{aligned}$$

Note that the second clause defining $\hat{a}$ can be resolved with the clause defining k_0' and the resulting program also has chain form. We would then have the clauses (4)–(7), for which we have already illustrated the top-down chart parser converted into an inference system.

5 An inference system for fixed-mode programs converted into chain form

5.1 Avoiding recomputation

The inference system we have so far, although sound and complete, only avoids the recomputation of a few partial results, when given a program resulting from our transformation. To illustrate this phenomenon, consider the following program for computing permutations. The predicate $perm(\langle L\rangle, \langle M\rangle)$ is intended to hold iff M is a permutation of the list L. This predicate uses an auxiliary predicate, $del(\langle L\rangle, \langle X, M\rangle)$, which is intended to hold iff M is the list L with the element X removed.

$$
\begin{aligned}
&perm(\langle L\rangle, \langle [X\,|\,Ys]\rangle) \leftarrow del(\langle L\rangle, \langle X, Xs\rangle),\ perm(\langle Xs\rangle, \langle Ys\rangle)\\
&perm(\langle [\,]\rangle, \langle [\,]\rangle) \leftarrow\\
\\
&del(\langle [X|Xs]\rangle, \langle Y, [X\,|\,Ys]\rangle) \leftarrow del(\langle Xs\rangle, \langle Y, Ys\rangle)\\
&del(\langle [X|Xs]\rangle, \langle X, Xs\rangle) \leftarrow
\end{aligned}
$$

The chain form of this program appears in Appendix C.

When computing the permutations of a 4-element list $[a, b, c, d]$, say, some partial results are computed many times. The reason is that only edges having exactly the same state (including the stack) are considered equal. For instance, among the created edges, there are two edges:

$$
\begin{aligned}
&edge(\langle \overbrace{[a,b]}^{\text{stack}}, [c,d]\rangle, \langle \overbrace{[a,b]}^{\text{stack}}, [c,d]\rangle, \widehat{perm}, [\,]) \text{ and}\\
&edge(\langle \underbrace{[b,a]}_{\text{stack}}, [c,d]\rangle, \langle \underbrace{[b,a]}_{\text{stack}}, [c,d]\rangle, \widehat{perm}, [\,]),
\end{aligned}
$$

that are syntactically identical to each other, except for the terms representing stacks, i.e. the first element in the first brackets and the first element in the second brackets. (Both edges assert that $[c, d]$ is a permutation of $[c, d]$.)

We will therefore make some changes to td-inference. Observe that directed programs converted into chain form have, by definition, the property that predicates with symbols $\hat{p}$ hold only over pairs of states having the same stack. We will call this the *constant-stack property*.

We can exploit the constant-stack property by repacing the fundamental rule by:

$$
\frac{edge(\langle St_1|\mathbf{x}\rangle, \langle St_2|\mathbf{y}\rangle, \hat{p}_0, [\hat{p}_i, k_i, \ldots, \hat{p}_n, k_n]) \qquad edge(\langle St_3|\mathbf{y}\rangle, \langle St_3|\mathbf{z}\rangle, \hat{p}_i, [\,])}{edge(\langle St_1|\mathbf{x}\rangle, \langle St_2|\mathbf{z}\rangle, \hat{p}_0, [k_i, \ldots, \hat{p}_n, k_n])}
$$

Thus, the partial result that was computed with stack St_3 is used by the edge having stacks St_1 and St_2.

We must keep, however, the original fundamental rule for the predicates with symbols of the form k_i:

$$\frac{edge(\langle St_1|\mathbf{x}\rangle, \langle St_2|\mathbf{y}\rangle, \hat{p}_0, [k_i, \hat{p}_{i+1}, \ldots, k_n]) \qquad edge(\langle St_2|\mathbf{y}\rangle, \langle St_3|\mathbf{z}\rangle, k_i, [\,])}{edge(\langle St_1|\mathbf{x}\rangle, \langle St_3|\mathbf{z}\rangle, \hat{p}_0, [\hat{p}_{i+1}, \ldots, k_n])}$$

With the above change, edges can use partial results having a different stack, but we still need some way of avoiding the recomputation of such results. We can achieve this by modifying the membership test. Now, whenever an edge of the form:

$$edge(\langle \mathbf{st}_1|\mathbf{x}\rangle, \langle \mathbf{st}_1|\mathbf{x}\rangle, \hat{p}_0, [k_0, \hat{p}_1, \ldots, \hat{p}_n, k_n])$$

is removed from the agenda, it is combined with the edges in the chart only if there is no edge of the form:

$$edge(\langle \mathbf{st}_2|\mathbf{x}\rangle, \langle \mathbf{st}_2|\mathbf{x}\rangle, \hat{p}_0, [k_0, \hat{p}_1, \ldots, \hat{p}_n, k_n])$$

in the chart. In addition, we can ignore an edge labeled with $(\hat{p}_0, [\,])$ removed from the agenda if there is a similar edge in the chart already, having a different stack. If an edge of the form:

$$edge(\langle \mathbf{st}_1|\mathbf{x}\rangle, \langle \mathbf{st}_1|\mathbf{y}\rangle, \hat{p}_0, [\,])$$

is removed from the agenda, it is combined with the edges in the chart only if there is no edge of the form:

$$edge(\langle \mathbf{st}_2|\mathbf{x}\rangle, \langle \mathbf{st}_2|\mathbf{y}\rangle, \hat{p}_0, [\,])$$

in the chart.

We will call this doctored system $\widehat{td}$*-inference*. Using the constant-stack property we can prove that $\widehat{\text{td}}$-inference is both sound and complete in the same way as td-inference is.

Appendix D contains the code of a proof procedure based on the above changes. This proof procedure computes each permutation of each sublist *exactly once* when applied to the *perm* program. By contrast, Prolog may compute some partial results for the *perm* program many times. When given the goal $\leftarrow perm(\langle [a,b,c,d]\rangle, \langle Zs\rangle)$, for instance, Prolog computes *six* times the (only) permutation of each singleton list: $[a]$, $[b]$, $[c]$, and $[d]$.

The proof procedure in Appendix D also avoids recomputation in the following program for solving the Towers of Hanoi problem:

$$\begin{array}{l} h(1, A, B, C, [\mathit{from}, A, \mathit{to}, B]) \leftarrow \\ h(N, A, B, C, \mathit{Moves}) \leftarrow N > 1,\ N' \ \mathit{is}\ N - 1, \\ \qquad\qquad\qquad\qquad\qquad h(N', A, C, B, L), \\ \qquad\qquad\qquad\qquad\qquad h(N', C, B, A, M), \mathit{append}(L, [\mathit{from}, A, \mathit{to}, B|M], \mathit{Moves}) \end{array}$$

For instance, moving 4 disks from peg a to peg b using auxiliary peg c, requires twice the result of moving 2 disks from peg a to peg b using auxiliary peg c. The program in Appendix D computes this result only once. In fact, the number of edges produced by this program is linear in the number of disks.

The same phenomenon occurs in a doubly recursive, naive definition of Fibonacci numbers:

$$
\begin{aligned}
& \mathit{fib}(0,1) \leftarrow \\
& \mathit{fib}(1,1) \leftarrow \\
& \mathit{fib}(N,F) \leftarrow N > 1, \quad N_1 \ \mathit{is}\ N-1, \quad \mathit{fib}(N_1,F_1), \\
& \qquad\qquad\qquad\qquad N_2 \ \mathit{is}\ N-2, \quad \mathit{fib}(N_2,F_2), \quad F \ \mathit{is}\ F_1+F_2
\end{aligned}
$$

in which our method uses memory space linear in the range of the Fibonacci number computed.

5.2 Completeness for datalog programs

We will now observe that $\widehat{\mathrm{td}}$-inference is complete for logic programs having no function symbols other than constants (i.e. datalog programs), in the sense that the search space has no infinite branches.

First, we remark that directed, datalog programs are transformed into chain programs which have function symbols other than constants only in the *first* component of the arguments of predicates (the added stack).

Next, we point out that in a chart created with $\widehat{\mathrm{td}}$-inference for a directed, datalog program converted into chain form, there can only be finitely many edges of the form:

$$
\begin{aligned}
& \mathit{edge}(\langle \mathbf{st}|\mathbf{x}\rangle, \langle \mathbf{st}|\mathbf{y}\rangle, \hat{p}_i, [\,]) \ \text{ and} \\
& \mathit{edge}(\langle \mathbf{st}|\mathbf{x}\rangle, \langle \mathbf{st}|\mathbf{x}\rangle, \hat{p}_0, [k_0, \hat{p}_1, \ldots, \hat{p}_n, k_n])
\end{aligned}
$$

due to the membership tests.

In addition, there can be at most one edge of the form:

$$\mathit{edge}(\langle \mathbf{st}_1|\mathbf{x}\rangle, \langle \mathbf{st}_2|\mathbf{y}\rangle, k_i, [\,])$$

for a given $\langle \mathbf{st}_1|\mathbf{x}\rangle$ and k_i, because the transformation into chain form produces only one (unit) clause with predicate symbol k_i. Thus, the number of edges of the form:

$$\mathit{edge}(\langle \mathbf{st}|\mathbf{x}\rangle, \langle \mathbf{st}'|\mathbf{z}\rangle, \hat{p}_0, [\hat{p}_i, \ldots, \hat{p}_n, k_n]) \qquad\qquad i > 0$$

is also finite. This allows us to conclude that the chart created with $\widehat{\mathrm{td}}$-inference for a directed, datalog program converted into chain form is finite.

Theorem 5.1 *Let P be a directed program having no function symbols other than constants, and $G = \leftarrow s(\mathbf{x}, Z)$ a chain goal. Then the search space for $\widehat{\mathrm{td}}$-inference for $\hat{P} \cup \{\hat{G}\}$ has no infinite branches, where $\hat{P}$ and $\hat{G}$ are the chain versions of P and G, respectively.*

This property of $\widehat{\mathrm{td}}$-inference does not mean that our method is only applicable to datalog programs. Using our method for fixed-mode logic programs with function symbols other than constants may also be advantageous, as we saw in Subsection 5.1.

6 Comparisons with other methods

In this section, we compare our inference systems with several existing methods.

6.1 Pereira and Warren's Earley deduction

Pereira and Warren have extended [18] Earley's algorithm [8] to a proof procedure for logic programs, which they call "Earley deduction," and we shall now compare their work with ours. Their proof procedure has the advantage that it can be applied to any logic program.

Two rules produce new clauses; when none can be applied, the process terminates. Since chart parsers are a generalization of Earley's algorithm, we can give such rules using the chart-parsing terminology.

1. If the chart contains a clause C having a selected literal that unifies with a unit clause either in the chart or in the program, then create the resolvent of C with that unit clause. (This rule is the counterpart of the fundamental rule as well as the extension of the base.)

2. If the chart contains a clause having a selected literal that unifies with the head of a nonunit clause C in the program, with most general unifier θ, then create the clause $C\theta$. (This rule parallels the top-down rule of chart parsing.)

A new clause is added to the chart only if there is no clause already in the chart that subsumes the new one. Subsumption, however, is NP-complete [10].

Earley deduction terminates for some programs if subsumption is replaced by a test for syntactic equality. This change results in a proof procedure that can be faster than the original Earley deduction and our methods. Our proof procedures, however, are preferable to this variant of Earley deduction in programs for which our methods terminate but such a variant does not. We now exhibit one such example. Given the directed program:

$$p(\langle\rangle, X) \leftarrow p(\langle\rangle, f(X))$$

and a chart initialized with the clause $ans(Y) \leftarrow p(\langle\rangle, Y)$, Earley deduction with a syntactic-equality test instead of subsumption produces the infinite sequence:

$$\begin{array}{l} p(\langle\rangle, Y) \leftarrow p(\langle\rangle, f(Y)) \\ p(\langle\rangle, f(Y)) \leftarrow p(\langle\rangle, f(f(Y))) \\ \vdots \end{array}$$

With subsumption, Earley deduction does terminate for this example. Our methods, by contrast, do not require subsumption and yet also terminate.

6.2 Tamaki and Sato's OLDT resolution

In this subsection, we consider another proof procedure for logic programs that also uses a dynamic programming technique: OLDT resolution [25]. Our description is meant for comparing such a procedure with our method. Hence, our account of OLDT resolution is only a summary of that of its inventors. We refer the reader to [25] for a thorough description.

OLDT resolution can be viewed as an elaboration of memoing techniques in functional programming languages. The main difference with such techniques is that OLDT resolution takes into account the fact that one subgoal may have more than one answer. This is achieved, on the one hand, by recording in a list the answers to a subgoal A that have already been computed. On the other hand, this method keeps for each goal G having as leftmost subgoal an instance A' of A and occurring in the SLD tree being constructed, a sublist of such answers that have already been used by G. The SLD tree is constructed with the following process. The newly obtained answers for A are resolved with the goal G. This may cause new answers for A to be obtained, which are added to the list of answers for A. These two tasks are repeated until no new answers are generated.

Similar results to those for Earley deduction are obtained. First, OLDT resolution, unlike our method, is applicable to arbitrary logic programs. Second, OLDT resolution, just as Earley deduction, is accompanied by a subsumption test at each derivation step. Our method employs a syntactic-equality test instead. Inhibiting the subsumption test in OLDT resolution for some predicates may increase the possibility of entering into a nonterminating loop. For example, consider again the program:

$$p(\langle\rangle, X) \leftarrow p(\langle\rangle, f(X))$$

with the goal $\leftarrow p(\langle\rangle, Y)$. If we suppress the subsumption test on the subgoals of this program, OLDT resolution will enter a nonterminating loop.

6.3 Dietrich's extension tables

A method similar to OLDT resolution is Dietrich's "extension tables" [7, 9]. Like OLDT resolution, this method interleaves the task of generating new answers for a subgoal A with the task of resolving such answers with a goal having an instance A' of A as leftmost subgoal. Unlike OLDT resolution, this method compiles the original Prolog program into another Prolog program. In addition, the extension-tables method avoids having to record which answers for A have already been used by each goal with an instance A' of A. This is achieved by traversing the portion of the SLD tree that has been traversed already, every time that new answers are generated.

As with Earley deduction and OLDT resolution, replacing subsumption by syntactic equality in extension tables results in a method which enters a nonterminating loop for some directed programs for which our methods halt.

6.4 Magic sets

SLD resolution is sometimes said to be a *top-down* inference rule: the reasoning is backwards starting from the conclusions [14]. The magic-sets method, by contrast, uses *bottom-up* inference, in which the reasoning is forwards starting from the hypotheses. Using bottom-up inference in a straightforward way is accompanied by a disadvantage which renders this kind of inference unacceptably inefficient: many facts are derived that are irrelevant from the goal's point of view. The magic-sets method avoids this side-effect to a significant extent by transforming the original logic program before applying bottom-up inference. Bottom-up inference applied to the transformed program is then more focused on the answers relevant to the goal than the same inference applied to the original program.

Magic sets and related methods are the work of various researchers. Chang [5], and McKay and Shapiro [17] are early investigations. More highly developed methods were suggested by Bancilhon et al. [2], and Rohmer et al. [19]. These authors assume that the only function symbols occurring in the program to be transformed are constants. Later, Beeri and Ramakrishnan [4], and Saccà and Zaniolo [23] observed that the magic-sets method is also sound for logic programs with function symbols other than constants. A summary of magic sets and similar transformations can be found in [3].

The magic-sets version of a directed program. Let P be a directed program, and G a goal of the form $\leftarrow s(\mathbf{x}_0, Z)$, where $\mathbf{x}_0$ is a ground term, and Z a variable. We define $\mathcal{M}(P,G)$, the *magic-sets version* of P with respect to G, as:

1. For every clause in P of the form:

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n) \qquad n \geq 0$$

   $\mathcal{M}(P,G)$ has the clauses:

(a) one *modified rule*:

$$\tilde{p}_0(t_0, t'_n) \leftarrow m\_p_0(t_0),\ \tilde{p}_1(t'_0, t_1),\ \tilde{p}_2(t'_1, t_2),\ \ldots,\ \tilde{p}_n(t'_{n-1}, t_n)$$

(b) one *magic rule* for each subgoal:

$$\begin{aligned}
m_p_1(t'_0) &\leftarrow m_p_0(t_0)\\
m_p_2(t'_1) &\leftarrow m_p_0(t_0),\ \tilde{p}_1(t'_0, t_1)\\
m_p_3(t'_2) &\leftarrow m_p_0(t_0),\ \tilde{p}_1(t'_0, t_1),\ \tilde{p}_2(t'_1, t_2)\\
&\vdots\\
m_p_n(t'_{n-1}) &\leftarrow m_p_0(t_0),\ \tilde{p}_1(t'_0, t_1),\ \ldots,\ \tilde{p}_{n-1}(t'_{n-2}, t_{n-1})
\end{aligned}$$

2. $\mathcal{M}(P, G)$ has the clause:

$$m\_s(\mathbf{x}_0) \leftarrow$$

At first sight, it might seem that because of using bottom-up inference, the magic-sets method builds proof trees in a bottom-up way. This is not true, however. Recall that programs for both top-down and bottom-up chart parsing can be written in Prolog [11], thus using top-down inference at the meta-level, for building parse trees in either direction. Similarly, in the magic-sets method, the use at the meta-level of bottom-up inference, does not necessarily mean a bottom-up direction in building proof trees. In fact, magic sets does so in the opposite direction. To see this, we consider a chain program and establish a correspondence between facts inferred by the magic-sets method and edges [22]. We associate the fact:

$$\tilde{p}(\mathbf{x}, \mathbf{y}) \leftarrow$$

with the edge:

$$edge(\mathbf{x}, \mathbf{y}, p, [\,])$$

In addition, we associate the fact:

$$m\_p(\mathbf{x}) \leftarrow$$

with the edge:

$$edge(\mathbf{x}, \mathbf{x}, p, [\ldots])$$

Bottom-up inference using a modified rule amounts then to several applications of the fundamental rule, as illustrated in Figure 4 (where a chain clause is assumed; a general treatment is in [22]). The same inference using a magic rule corresponds to applications of the fundamental rule followed by an application of the *top-down* rule. With this correspondence between facts and edges, it is possible to logically derive $\mathcal{M}(P, G)$ from clauses describing the top-down chart parser together with an extension of P [22]. Hence, there is a close connection between the magic-sets method and td-inference.

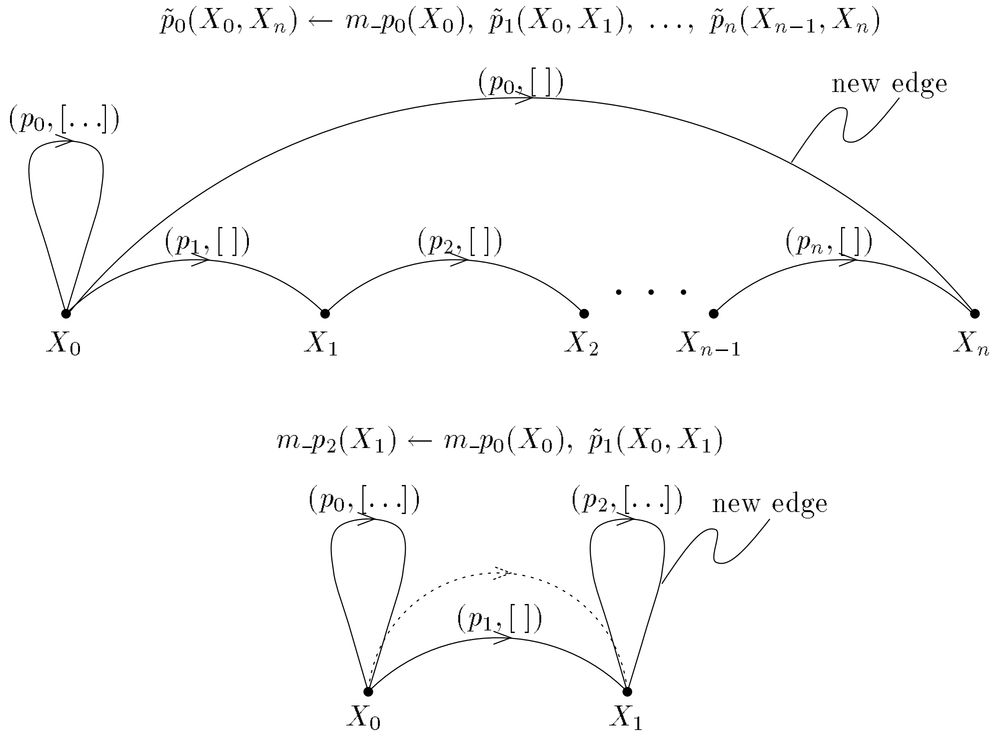

Figure 4: *Chart interpretation of clauses in* $\mathcal{M}(P, G)$.

6.5 Summary of comparisons

We gave a program causing Earley deduction, OLDT resolution, and extension tables enter a nonterminating loop if subsumption (which is a source of inefficiency) is eliminated. Of the methods we considered here, only magic sets and our mechanisms terminate for such a program without using subsumption. The magic-sets method, however, builds proof trees in a top-down manner, whereas we can do so in either direction, depending on the chart parser on which the inference system is based. Sometimes, it may be advantageous to build proof trees in a bottom-up way. Consider a program of the form:

$$
\begin{aligned}
&p(X_0, X_2) \leftarrow p(X_0, X_1),\ a_0(X_1, X_2)\\
&p([X|Xs], Xs) \leftarrow\\
&a_0([X|Xs], Xs) \leftarrow\\
&a_0(X_0, X_1) \leftarrow a_1(X_0, X_1)\\
&a_1(X_0, X_1) \leftarrow a_2(X_0, X_1)\\
&\vdots\\
&a_{n-1}(X_0, X_1) \leftarrow a_n(X_0, X_1)
\end{aligned}
$$

together with a goal of the form $\leftarrow p([1, 2, \ldots, m], Z)$. Both the magic-sets method and our method based on a top-down chart parser infer $n \cdot m$ irrelevant facts, none of which are inferred by our inference system based on a bottom-up chart parser. Conversely, it is easy to show a program for which magic sets and our method based on a top-down chart parser are preferable. This suggests an inference system based on a combination of both top-down and bottom-up chart parsing. (Haruno et al. [12] have combined both kinds of chart parsing in natural-language generation.) This possibility remains to be explored.

7 Concluding remarks

Chart parsers. We have seen that chart parsers can be readily converted into inference systems for a generalization of the difference-list representation of context-free grammars. This generalization keeps the nonunit clauses, but replaces unit clauses (which are of the form $p([c|X], X) \leftarrow$), by clauses having terms subject to only one syntactic restriction: all variables in the second argument must appear in the first argument (clauses of the form $p(t, t') \leftarrow$, where $\mathrm{var}(t') \subseteq \mathrm{var}(t)$). These programs are said to have chain form.

We proved both soundness and completeness of the inference system based on a top-down chart parser. An advantage of these systems over that of Prolog, for instance, is that some partial results are recorded, thus avoiding the recomputation of such results.

Conversion of fixed-mode programs into chain programs. It is possible to transform [20] fixed-mode logic programs into chain programs by adding arguments that play the role of a stack. Consequently, the inference systems obtained are also applicable to fixed-mode logic programs.

Fixed-mode logic programs. We believe that the class of fixed-mode logic programs, which encode procedural knowledge, is important. Although logic programming emphasizes a declarative reading, logic programs encoding procedural knowledge occur often in practice. Two reasons explain this. On the one hand, as observed by Kowalski [15], sometimes it is more natural to think procedurally first. On the other hand, there are obstacles to the invertibility of logic programs.

Chart parsers for transformed programs. A difficulty appears when using the obtained inference systems for programs resulting from our transformation into chain form. The stack added by this transformation makes these inference systems avoid the recomputation of partial results only in a few cases. The reason is that two partial results are considered to be the same one when two states (including their stacks) are the same. Thus, we modified the first version of the inference system based on a top-down chart parser so that the added stack is irrelevant for deciding whether a partial result has already been computed or not. We saw that the modified inference system not only avoids the recomputation of many partial results, but is complete for (fixed-mode) datalog programs (i.e. logic programs with no function symbols other than constants) in the sense that the search space has no infinite branches.

Although we concentrated on an inference system based on a top-down chart parser, the corresponding bottom-up variant can be obtained in a similar way.

Other methods. We observed that our methods may be advantageous over several existing inference systems; we considered Earley deduction, OLDT resolution, extension tables, and magic sets. The first three of these methods use subsumption, which is a source of inefficiency. If subsumption is eliminated, these methods may enter a nonterminating loop for programs for which magic sets and our inference systems halt without using subsumption.

The magic-sets method does not require subsumption, but builds proof trees in a top-down manner (even though it uses bottom-up inference). An advantage of our methods over magic sets is that we can build proof trees in either direction.

Bridge between programs and grammars. Our main contribution lies, we believe, in having exhibited a close connection between chain and fixed-mode logic programs on one side, with context-free grammars on the other side. We have elsewhere [21] adapted an LR parser from which we obtained another inference system for fixed-mode logic programs, using our transformation into chain form. We hope that these results suggest how to convert more parsers into inference systems for logic programs.

Acknowledgments

Special thanks to Julio Peralta and Mauricio Alvarez-Manilla, for many fruitful discussions. We are also grateful to Felipe Bracho, Carlos Brody, Warren Greiff, Rafael Ramírez, Carlos Silva, Paul Strooper, and Carlos Velarde, who gave valuable suggestions. The constructive criticisms of the referees were helpful in improving a previous version of this paper. We acknowledge the facilities provided by IIMAS, UNAM. This work was supported by grant IN301192 of DGAPA, UNAM.

Appendices

A Conversion of fixed-mode programs into chain form

Our method for converting fixed-mode programs to chain form is based on the following theorem.

Theorem A.1 *Let C be a directed clause:*

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n)$$

and let

$$\Pi_i = (\mathit{var}(t_0) \cup \cdots \cup \mathit{var}(t_{i-1})) \cap (\mathit{var}(t'_i) \cup \cdots \cup \mathit{var}(t'_n))$$

for $i = 1, \ldots, n$, and $\Pi_0 = \Pi_{n+1} = \emptyset$. Then the clause:

$$\hat{p}_0(U_0, U'_n) \leftarrow k_0(U_0, U'_0),\ \hat{p}_1(U'_0, U_1),\ k_1(U_1, U'_1),\ \hat{p}_2(U'_1, U_2),\ \ldots,$$
$$k_{n-1}(U_{n-1}, U'_{n-1}),\ \hat{p}_n(U'_{n-1}, U_n),\ k_n(U_n, U'_n)$$

is logically implied by C, the standard equality theory, the "iff" version of the function substitutivity axiom for the list-constructor function symbol, and the following axioms:

$$\hat{p}_i(\langle St|X\rangle, \langle St'|Y\rangle) \leftrightarrow St = St' \ \&\ p_i(X, Y) \qquad i = 0, \ldots, n$$
$$k_0(U, V) \leftrightarrow \exists Y_{1,0} \cdots \exists Y_{m_0,0}.[\langle \Sigma_0 | t_0\rangle = U \ \&\ \langle \Sigma_1 | t'_0\rangle = V]$$
$$k_1(U, V) \leftrightarrow \exists Y_{1,1} \cdots \exists Y_{m_1,1}.[\langle \Sigma_1 | t_1\rangle = U \ \&\ \langle \Sigma_2 | t'_1\rangle = V]$$
$$\vdots$$
$$k_n(U, V) \leftrightarrow \exists Y_{1,n} \cdots \exists Y_{m_n,n}.[\langle \Sigma_n | t_n\rangle = U \ \&\ \langle \Sigma_{n+1} | t'_n\rangle = V]$$

where $Y_{1,i}, \ldots, Y_{m_i,i}$ are the variables on the right-hand side of the definition of k_i, except for U and V, for $i = 0, \ldots, n$; Σ_i is any list of the form $[X_{1,i}, \ldots, X_{d_i,i}|St]$, $\{X_{1,i}, \ldots, X_{d_i,i}\} = \Pi_i$, if $\Pi_i \neq \emptyset$, and Σ_i is St if $\Pi_i = \emptyset$, for $i = 0, \ldots, n$.

Sketch proof *Stage 1.* It is possible to construct a partial derivation with initial goal $\leftarrow p_0(U, V)$ using the standard equality theory such that the following clause C_1 is a conditional answer:

$$p_0(t_0, s'_n) \leftarrow X_{1,1} = Z_{1,1},\ \ldots,\ X_{m_1,1} = Z_{m_1,1},\ \ldots,$$
$$X_{1,n} = Z_{1,n},\ \ldots,\ X_{m_n,n} = Z_{m_n,n},$$
$$p_1(s'_0, t_1),\ p_2(s'_1, t_2),\ \ldots,\ p_n(s'_{n-1}, t_n)$$

where $\{X_{1,i}, \ldots, X_{m_i,i}\} = (\mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_{i-1})) \cap \mathrm{var}(t'_i)$, for $i = 1, \ldots, n$; s'_i is $t'_i\{X_{1,i}/Z_{1,i}, \ldots, X_{m_i,i}/Z_{m_i,i}\}$ for $i = 1, \ldots, n$; s'_0 is t'_0; and $Z_{1,1}, \ldots, Z_{m_n,n}$ are distinct variables not occurring in C. This conditional answer can be obtained by disassembling the t'_is using function substitutivity until the equations in C_1 are produced. The s'_is can then be assembled using reflexivity.

Stage 2. Each equation in C_1 is of the form $X_{j,k} = Z_{j,k}$, where $X_{j,k}$ occurs in exactly one t_i for $i < k$, and $Z_{j,k} \in s'_k$. Using transitivity, we can derive a clause C_2 equal to C_1, except that instead of each equation $X_{j,k} = Z_{j,k}$, where $X_{j,k} \in t_i$, C_2 has the equations:

$$X_{j,k} = Y_{i+1},\ Y_{i+1} = Y_{i+2},\ \ldots,\ Y_{k-1} = Z_{j,k}$$

where $Y_{i+1}, \ldots, Y_{k-1}$ are distinct variables not occurring in C_1. Next, from C_2, transitivity, and the definition of $\hat{p}_0$, we can derive the clause C_3:

$$\begin{aligned}\hat{p}_0(\langle St_0|t_0\rangle, \langle St_n|s'_n\rangle) \leftarrow\ & St_0 = St_1,\ St_1 = St_2,\ \ldots,\ St_{n-1} = St_n,\\ & \ldots,\ X_{j,k} = Y_{i+1},\ Y_{i+1} = Y_{i+2},\ \ldots,\ Y_{k-1} = Z_{j,k},\ \ldots,\\ & p_1(s'_0, t_1),\ p_2(s'_1, t_2),\ \ldots,\ p_n(s'_n, t_n)\end{aligned}$$

where $X_{j,k} \in \mathrm{var}(t_i)$ and $Z_{j,k} \in \mathrm{var}(s'_k)$. Folding now the "iff" version $[U|V] = [U'|V'] \leftrightarrow U = U'\ \&\ V = V'$ of the function substitutivity axiom for the list-constructor function symbol, and the definitions of $\hat{p}_l$ for $l = 1, \ldots, n$, we get C_4:

$$\hat{p}_0(\hat{t}_0, \hat{t}'_n) \leftarrow \hat{p}_1(\hat{t}'_0, \hat{t}_1),\ \hat{p}_2(\hat{t}'_1, \hat{t}_2),\ \ldots,\ \hat{p}_n(\hat{t}'_{n-1}, \hat{t}_n)$$

where:

1. $\mathrm{var}(\hat{t}_i) \cap \mathrm{var}(\hat{t}_j) = \emptyset$, for $i, j = 0, \ldots, n$ and $i \neq j$;
2. $\mathrm{var}(\hat{t}'_i) \subseteq \mathrm{var}(\hat{t}_i)$, for $i = 0, \ldots, n$.

The soundness of these fold steps can be proved using, for instance, the result in [24, pages 57, 58].

Stage 3. Finally, we can fold the definitions of k_i for $i = 0, \ldots, n$ in the clause:

$$\begin{aligned}\hat{p}_0(U_0, U'_n) \leftarrow\ & \hat{t}_0 = U_0,\ \hat{t}'_n = U'_n,\\ & \hat{t}'_0 = U'_0,\ \hat{t}_1 = U_1,\ \ldots,\\ & \hat{t}'_{n-1} = U'_{n-1},\ \hat{t}_n = U_n,\\ & \hat{p}_1(U'_0, U_1),\ \hat{p}_2(U'_1, U_2),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n)\end{aligned}$$

which can be obtained as a conditional answer from the goal $\leftarrow \hat{p}_0(U_0, U'_n)$ using predicate substitutivity. These fold steps can also be proved to be sound using the result in [24, pages 57, 58]. The final clause has the claimed form. Hence the theorem holds. □

B Proof of Lemma 3.2

We give a proof sketch of an auxiliary lemma for proving completeness of td-inference (Theorem 3.3); a more detailed proof of this lemma is in [1].

Lemma 3.2 *Let P be a chain program. If $P \models q(\mathbf{x}, \mathbf{y})$, then there exists a td-derivation for P and* $\{\mathrm{edge}(\mathbf{w}, \mathbf{x}, p, [q, \ldots])\}$ *ending in a chart containing* $\mathrm{edge}(\mathbf{x}, \mathbf{y}, q, [\,])$*, where* $\mathbf{w}$*,* $\mathbf{x}$*, and* $\mathbf{y}$ *are ground terms.*

Proof sketch By the completeness of SLD resolution [16, page 49], we can assume that there exists an SLD refutation for $P \cup \{\leftarrow q(\mathbf{x}, \mathbf{y})\}$. The proof is by induction on the length l of such an SLD refutation.

Basis: If $l = 1$, then there exists a clause $(q(t, t') \leftarrow) \in P$ such that $\mathbf{x}$ and t unify with substitution θ. In terms of td-inference, we can apply the base-extension rule using $edge(\mathbf{w}, \mathbf{x}, p, [q, \ldots])$ and $q(t, t') \leftarrow$, to infer $edge(\mathbf{x}, \mathbf{y}, q, [\,])$.

Inductive step: Next, suppose that the result holds for SLD refutations of length less than l. By the independence of the computation rule [16, page 51], we can assume a refutation for $P \cup \{\leftarrow q(\mathbf{x}, \mathbf{y})\}$ which selects the leftmost subgoal of each goal. Let the first program clause used in such a refutation be:

$$q(X_0, X_n) \leftarrow q_1(X_0, X_1),\ q_2(X_1, X_2),\ \ldots,\ q_n(X_{n-1}, X_n) \tag{13}$$

Then, by the lifting lemma [16, page 48], there exists an SLD refutation for $P \cup \{\leftarrow q_1(\mathbf{x}, X_1)\}$. Let θ_1 be the answer substitution of such a refutation. First, we apply the top-down rule to $edge(\mathbf{w}, \mathbf{x}, p, [q, \ldots])$ and (13) obtaining:

$$edge(\mathbf{x}, \mathbf{x}, q, [q_1, q_2, \ldots, q_n]) \tag{E_1}$$

By the induction hypothesis, using (E_1), there exists a td-derivation for P and a chart having (E_1), which ends in a chart containing:

$$edge(\mathbf{x}, X_1\theta_1, q_1, [\,]) \tag{E_1'}$$

Next, applying the fundamental rule to (E_1) and (E_1'), we obtain:

$$edge(\mathbf{x}, X_1\theta_1, q, [q_2, \ldots, q_n]) \tag{E_2}$$

Again, by the induction hypothesis, using now (E_2), there exists a td-derivation for P and a chart having (E_2), which ends in a chart containing:

$$edge(X_1\theta_1, X_2\theta_2, q_2, [\,]) \tag{E_2'}$$

Applying the fundamental rule, this time to (E_2) and (E_2'), we infer:

$$edge(\mathbf{x}, X_2\theta_2, q, [q_3, \ldots, q_n]) \tag{E_3}$$

Alternatively applying the induction hypothesis and the fundamental rule $n-2$ times more, we conclude there exists a td-derivation for P and a chart having (E_i), which ends in a chart containing edges of the form:

$$edge(X_{i-1}\theta_{i-1}, X_i\theta_i, q_i, [\,]) \tag{E_i'}$$
$$edge(\mathbf{x}, X_i\theta_i, q, [q_{i+1}, \ldots, q_n]) \tag{E_{i+1}}$$

for $i = 3, \ldots, n$.

By choosing each θ_i appropriately, it is possible to make $X_n\theta_n = \mathbf{y}$, so that (E_{n+1}) is $edge(\mathbf{x}, \mathbf{y}, q, [\,])$. We conclude that the lemma holds. $\square$

C Other programs converted into chain form

In this appendix, we give two more examples of directed programs converted into chain form with our method. The directed version of these programs appeared in Section 5.

Permutations. The predicate $perm(\langle L\rangle, \langle M\rangle)$ is intended to hold iff M is a permutation of the list L. This predicate uses an auxiliary predicate, $del(\langle L\rangle, \langle X, M\rangle)$, which is intended to hold iff M is the list L with the element X removed.

The directed program is:

$$perm(\langle L\rangle, \langle [X\,|\,Ys]\rangle) \leftarrow del(\langle L\rangle, \langle X, Xs\rangle),\ perm(\langle Xs\rangle, \langle Ys\rangle)$$
$$perm(\langle [\,]\rangle, \langle [\,]\rangle) \leftarrow$$

$$del(\langle [X|Xs]\rangle, \langle Y, [X\,|\,Ys]\rangle) \leftarrow del(\langle Xs\rangle, \langle Y, Ys\rangle)$$
$$del(\langle [X|Xs]\rangle, \langle X, Xs\rangle) \leftarrow$$

The corresponding program in chain form is:

$$\widehat{perm}(U_0, U_5) \leftarrow k_0(U_0, U_1),\ \widehat{del}(U_1, U_2),\ k_1(U_2, U_3), \widehat{perm}(U_3, U_4),\ k_2(U_4, U_5)$$
$$k_0(\langle St, L\rangle, \langle St, L\rangle) \leftarrow$$
$$k_1(\langle St, X, Xs\rangle, \langle [X|St], Xs\rangle) \leftarrow$$
$$k_2(\langle [X|St],\ Ys\rangle, \langle St, [X\,|\,Ys]\rangle) \leftarrow$$

$$\widehat{perm}(U_0, U_1) \leftarrow k_0'(U_0, U_1)$$
$$k_0'(\langle St, [\,]\rangle, \langle St, [\,]\rangle) \leftarrow$$

$$\widehat{del}(U_0, U_3) \leftarrow c_1(U_0, U_1),\ \widehat{del}(U_1, U_2),\ c_2(U_2, U_3)$$
$$c_1(\langle St, [X|Xs]\rangle, \langle [X|St], Xs\rangle) \leftarrow$$
$$c_2(\langle [X|St], Y, Ys\rangle, \langle St, Y, [X\,|\,Ys]\rangle) \leftarrow$$

$$\widehat{del}(U_0, U_1) \leftarrow c_0'(U_0, U_1)$$
$$c_0'(\langle St, [X|Xs]\rangle, \langle St, X, Xs\rangle) \leftarrow$$

Towers of Hanoi. The predicate $h(N, A, B, C, Moves)$ is intended to hold iff $Moves$ is the result of moving N disks from peg A to peg B using auxiliary peg C.

$$
\begin{aligned}
&h(1, A, B, C, [from, A, to, B]) \leftarrow \\
&h(N, A, B, C, Moves) \leftarrow N > 1,\ N' \ is\ N - 1, \\
&\qquad\qquad\qquad\qquad\quad h(N', A, C, B, L), \\
&\qquad\qquad\qquad\qquad\quad h(N', C, B, A, M),\ append(L, [from, A, to, B|M], Moves)
\end{aligned}
$$

This example illustrates how to use Prolog's built-in arithmetic, as well as Prolog's definition of "*append*": we have incorporated certain subgoals into the body of clauses defining some k_i's.

$$
\begin{aligned}
&\hat{h}(\langle St, 1, A, B, C\rangle, \langle St, [from, A, to, B]\rangle) \leftarrow \\
&\hat{h}(U_0, U_5) \leftarrow k_0(U_0, U_1),\ \hat{h}(U_1, U_2),\ k_1(U_2, U_3),\ \hat{h}(U_3, U_4),\ k_2(U_4, U_5) \\
&k_0(\langle St, N, A, B, C\rangle, \langle [N', A, B, C|St], N', A, C, B\rangle) \leftarrow N > 1,\ N' \ is\ N - 1 \\
&k_1(\langle [N, A, B, C|St], L\rangle, \langle [A, B, L|St], N, C, B, A\rangle) \leftarrow \\
&k_2(\langle [A, B, L|St], M\rangle, \langle St, Moves\rangle) \leftarrow append(L, [from, A, to, B|M], Moves)
\end{aligned}
$$

D A proof procedure based on a top-down chart parser

Next, we exhibit the Prolog top-down chart parser of [11, pages 208, 211, and 212], modified as explained in Section 5.

```
% ?- test( [[],[1,2]] ). example goal
test( State ) :-
     initial( Symbol ),
     unit_clauses( Symbol, State, Agenda1 ),
     start_active( Symbol, State, Agenda2 ),
     append( Agenda1, Agenda2, Agenda12 ),
     extend_base_edges( Agenda12, Agenda3 ),
     append( Agenda12, Agenda3, Agenda ),
     extend_edges( Agenda, [], Chart2 ).

unit_clauses( Symbol, State, Edges ) :-
     findall( edge(State,State1,Symbol,[]),
              word( Symbol, State, State1 ),
              Edges ).
```

```
start_active( Category, State, Edges1 ) :-
     findall( edge( State, State, Category, Categories ),
              rule( Category, Categories ),
              Edges ),
     advance_ks( Edges, Edges1 ).

extend_base_edges( Edges, NewBaseEdges ) :-
     findall( NewBaseEdge,
              ext_base(Edges,NewBaseEdge),
              NewBaseEdges ).

ext_base( Edges1, edge(Y,Z,Q,[]) ) :-
     member( edge(X,Y,P,[Q|_]), Edges1 ),
     word( Q, Y, Z ).

advance_ks( [], [] ).
advance_ks( [edge(X,Y,H,[K|Cats])|Edges], [edge(X,Z,H,Cats)|Edges1] ) :-
     word( K, Y, Z ), !,
     advance_ks( Edges, Edges1 ).
advance_ks( [edge(X,Y,H,[K|Cats])|Edges], Edges1 ) :-
     advance_ks( Edges, Edges1 ).

extend_edges( [], Chart, Chart ).
extend_edges( [edge([St1|X],[St2|Y],H,[])|Agenda1], Chart1, Chart2 ) :-
     member( edge([_|X],[_|Y],H,[]), Chart1 ), !,
     extend_edges( Agenda1, Chart1, Chart2 ).
extend_edges( [edge([St1|X],[St2|Y],H,Cats)|Agenda1], Chart1, Chart2 ) :-
     rule( H, [_|Cats] ), %empty active edge
     member( edge([_|X],[_|Y],H,Cats), Chart1 ), !,
     extend_edges( Agenda1, Chart1, Chart2 ).
extend_edges( [Edge|Agenda1], Chart1, Chart3 ) :-
     Chart2 = [Edge|Chart1],
     new_edges( Edge, Chart2, Edges ),
%    add_edges( Agenda1, Edges, Agenda2 ), % breadth-first
     add_edges( Edges, Agenda1, Agenda2 ), % depth-first
     extend_edges( Agenda2, Chart2, Chart3 ).

add_edge( Edge, Edges, Edges ) :- member( Edge, Edges ), !.
add_edge( Edge, Edges, [Edge|Edges] ).
```

```
add_edges( [], Edges, Edges ).
add_edges( [Edge|Edges], Edges1, Edges3 ) :-
     add_edge( Edge, Edges1, Edges2 ),
     add_edges( Edges, Edges2, Edges3 ).

new_edges( edge([_|Y],[_|Z],Category1,[]), Chart, Edges ) :-
     findall( edge([St0|X],[St1|Z],Category2,Categories),
              member(edge([St0|X],[St1|Y],Category2,
                                           [Category1|Categories]),
                     Chart),
              Edges1 ), %fundamental rule; passive edge was added
     advance_ks( Edges1, Edges11 ),
     extend_base_edges( Edges11, Edges2 ), %extend base
     add_edges( Edges11, Edges2, Edges ).
new_edges( edge([St1|X],[St2|Y],Category1,[Category2|Categories]),
                                                        Chart, Edges ) :-
     start_active( Category2, [St2|Y], Edges1 ), %top-down rule
     findall( edge([St1|X],[St2|Z],Category1,Categories),
              member(edge([_|Y],[_|Z],Category2,[]), Chart),
              Edges2 ), %fundamental rule; active edge was added
     advance_ks( Edges2, Edges21 ),
     append( Edges1, Edges21, Edges12 ),
     extend_base_edges( Edges12, Edges3 ), %extend base
     add_edges( Edges12, Edges3, Edges ).

rule( a, [k0,a,k1] ).

word( a, [St,Z], [St,[],Z] ).
word( k0, [St,[W|Z]], [[W|St],Z] ).
word( k1, [[W|St],X,Y], [St,[W|X],Y] ).

initial( a ).
```

References

[1] Mauricio Alvarez-Manilla. *Programas Orientados a Estados: Su Relación con Programas Lógicos y Lenguajes Formales.* Universidad Nacional Autónoma de México, 1995. B.Sc. thesis.

[2] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth*

ACM Symposium on Principles of Database Systems, pages 1–15, 1986.

[3] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. Technical Report 772, Computer Sciences Department, University of Wisconsin, Madison, June 1988.

[4] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 269–283, 1987.

[5] C.L. Chang. On the evaluation of queries containing derived relations in relational databases. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory*, volume I, pages 235–260. Plenum Press, 1981.

[6] Keith L. Clark and M.H. van Emden. Consequence verification of flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–60, January 1981.

[7] Suzanne Wagner Dietrich. Extension tables: memo relations in logic programming. In *Proceedings of the IEEE 1987 Symposium on Logic Programming*, pages 264–272, San Francisco, CA, 1987.

[8] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[9] Changguan Fan and Suzanne Wagner Dietrich. Extension table built-ins for Prolog. *Software—Practice and Experience*, 22(7):573–597, 1992.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog. An Introduction to Computational Linguistics*. Addison-Wesley, 1989.

[12] Masahiko Haruno, Yasuharu Den, Yuji Matsumoto, and Makoto Nagao. Bidirectional chart generation of natural language texts. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 350–356, 1993.

[13] Martin Kay. Algorithm schemata and data structures in syntactic processing. Technical Report CSL-80-12, Xerox Palo Alto Research Center, October 1980.

[14] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–436, 1979.

[15] Robert Kowalski. Logic programming principles. In Anthony Ralston and Edwin D. Reilly, editors, *Encyclopedia of Computer Science*, pages 778–783. Van Nostrand Reinhold, 3rd edition, 1993.

[16] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[17] Donald P. McKay and Stuart C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, 1981.

[18] Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, 1983. Association for Computational Linguistics.

[19] J. Rohmer, R. Lescoeur, and J.M. Kerisit. The Alexander method —A technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4:273–285, 1986.

[20] David A. Rosenblueth. Fixed-mode logic programs as state-oriented programs. Technical Report Preimpreso No. 2, IIMAS, UNAM, 1991.

[21] David A. Rosenblueth and Julio C. Peralta. LR inference: Inference systems for fixed-mode logic programs, based on LR parsing. In Maurice Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium*, pages 439–453, Ithaca, N.Y., U.S.A., 1994.

[22] David A. Rosenblueth, Julio C. Peralta, and Carlos Silva S. A grammatical reconstruction of the magic-sets method and an alternative method based on a parser. Technical Report Preimpreso No. 12, IIMAS, UNAM, 1993.

[23] Domenico Saccà and Carlo Zaniolo. Implementation of recursive queries for a data language based on pure Horn logic. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Logic Programming Conference*, pages 104–135, Melbourne, Australia, 1987. MIT Press.

[24] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[25] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *Proceedings of the Third International Logic Programming Conference*, pages 84–98. Springer-Verlag Lecture Notes in Computer Science 225, 1986.

[26] Phil Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432. Springer-Verlag Lecture Notes in Computer Science 225, 1986.