PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Chart Parsers as Proof Procedures for Fixed-Mode Logic Programs

David A. Rosenblueth

IIMAS, UNAM
Apdo. 20-726, 01000 Mexico D.F.
drosenbl@unamvm1.bitnet

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. Just as there are parsers with advantages over that simple one, it may be desirable to develop other proof procedures for logic programs than the one used by Prolog. The similarity between definite clauses and productions suggests looking at parsing to develop such procedures. We show that for an important class of logic programs (fixed-mode logic programs with ground data structures) the conversion of parsers into proof procedures can be straightforward. This allows for proof procedures that construct refutations that Prolog does not find and opens up opportunities for parallelism.

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 found numerous applications, its execution mechanism has several disadvantages. For instance, if such a mechanism finds an infinite branch of the derivation tree, it enters a nonterminating loop. Thus, it may be desirable to develop new proof procedures for logic programs.

Simple parsers with backtracking also enter nonterminating loops easily. This has motivated the design of other more sophisticated parsing methods. In contrast with proof procedures for logic programs, there already exists a great variety of parsers. The resemblance between definite clauses and productions suggests looking at parsers to develop new proof procedures.

Pereira and Warren [1983] have adapted Earley's [1970] parsing algorithm, but the result is inefficient compared with Prolog. It uses subsumption, which is NP-complete [Garey and Johnson 1979]. We show that by considering a restricted class of logic programs, parsers can be readily adapted to proof procedures. This class is important: it consists of fixed-mode logic programs with ground data structures. Moreover, our proof procedures do not use subsumption and may be more efficient than Pereira and Warren's.

Compositional 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 do so by allowing the body of clauses to denote the composition of arbitrary binary relations; we call such programs "compositional." Prolog programs are not normally written in compositional form. Thus, we consider programs in a larger class (fixed-mode programs with ground data structures) and transform [Rosenblueth 1991] them into compositional 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 query. When we refer to a "fixed-mode logic program," we actually mean a program and a query such that with Prolog's computation rule all subgoals with the same predicate symbol have the same mode. By further restricting these programs to have "ground data structures," we require all arguments in a subgoal that are not variables to be ground terms when the subgoal is selected. This class of program is important because it includes many programs occurring in practice.

At first glance, it seems that the presence of difference lists causes a program to have data structures with variables. However, by separating both components of a difference list it is possible to write some programs using

difference lists as programs with ground data structures. (The usual quicksort program is such an example; the sorted list is then built backwards.)

Overview of the paper. The rest of this paper is organized as follows. Section 2 reviews chart parsers. Section 3 shows that such parsers are also correct for compositional programs. Section 4 deals with a method for converting fixed-mode to compositional programs, thus making chart parsers proof procedures for the former class of programs. Section 5 compares these procedures with Pereira and Warren's. Section 6 concludes this paper with some remarks.

2 Chart parsers

Charts. Chart parsers [Gazdar and Mellish 1989] are methods for parsing strings of context-free languages that can be regarded as a generalization of Earley's algorithm. A *chart* is a set of "partially" applied productions, usually called *edges*. Each edge contains, in addition to the part of a production to be applied and the left-hand side of that production, two pointers to symbols of the string being parsed. The substring between these pointers corresponds to the part of that production that has already been applied.

It is useful to classify edges into those that have not been applied at all: *empty active edges*, those that have already been applied completely: *passive edges*, and all the others: *nonempty active edges*.

The fundamental rule. New edges are created according to the following rule, often called the *fundamental rule*.

If a chart contains:

1. *an active edge (either empty or nonempty) from point* a *to point* b *in which the next symbol to be applied is* Q*, and*
2. *a passive edge with left-hand side* Q*, from point* b *to point* c,

then create a new edge from a *to* c *in which the production is the same as the one in the active edge, but with* Q *applied.* Figure 1 illustrates this rule. In figures representing edges, we use the following notation. Each edge is labeled with an arrow, a symbol to the left of the arrow, and a possibly empty string to the right. The symbol is the left-hand side of the partially applied production. The string is the part of that production that remains to be applied.

Top-down and bottom-up parsing. The fundamental rule takes only existing edges to create new ones, and does not use information from the set of productions.

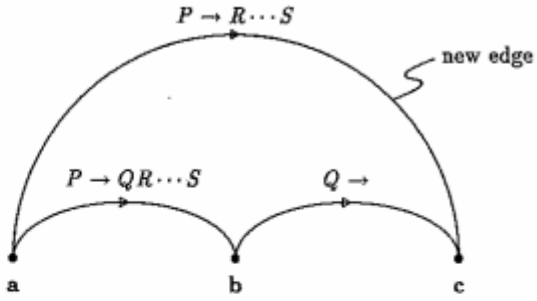

Figure 1: *The fundamental rule.*

Therefore, a mechanism is needed for building edges from productions. Two main mechanisms for this purpose are used, commonly called "top-down" and "bottom-up" rules. The former builds parse trees from the root towards the leaves, and the latter does so from the leaves towards the root.

The *top-down rule* creates edges as follows. *If an active edge from* a *to* b *is added to the chart, in which the next symbol to be applied is* Q*, then create one empty active edge from* b *to* b *for every production having* Q *as left-hand side and labeled with that production.* Figure 2 exemplifies this rule.

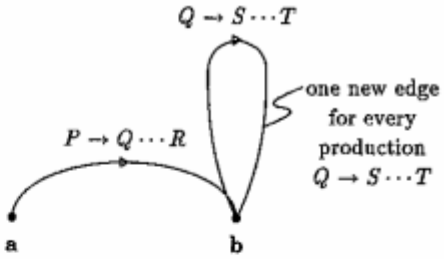

Figure 2: *The top-down rule.*

Given a parse tree having a leaf Q and a node P as parent of Q, this rule allows for Q to be expanded by creating an empty active edge with Q as left-hand side. Hence, parse trees are built by expanding the leaves with nonterminals, which is a construction of parse trees from the root towards the leaves.

The *bottom-up rule* creates edges as follows. *If a passive edge from* a *to* b *is added to the chart, in which the left-hand side symbol is* Q*, then create one empty active edge from* a *to* a *for every production having* Q *as first symbol on the right-hand side and labeled with that production.* This rule is depicted in Figure 3.

The bottom-up rule takes a passive edge, representing a parse subtree with Q as root. By creating an empty active edge with Q as first symbol to be applied, and P as left-hand side, Q becomes the child of a node P, which is the root of a new subtree. Thus, this rule builds parse trees from the leaves towards the root.

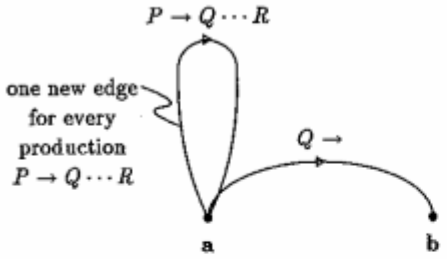

Figure 3: *The bottom-up rule.*

Base of the chart. The fundamental rule takes two edges. One of them is active and the other one passive. The next symbol to be applied in the former must be the left-hand side of the latter. This means that the case where the next symbol to be applied is a terminal is not covered (all left-hand sides of productions are nonterminals). We can remedy this situation by assuming that the productions have been written in such a way that each terminal occurs only in productions with exactly one symbol (that terminal) on the right-hand side. Now we can create certain edges as follows. For each production with a terminal occurring in the string being parsed, we create a passive edge from that terminal to the next one, labeled with that production. We can do so, because an edge represents a partially applied production (where "partially" may mean "completely") and all those productions can be immediately applied. Now we can rely only on the fundamental rule to operate existing edges. We shall call the set of all edges created from terminals the *base* of the chart.

Initialization. To initialize a parser using the bottom-up rule, it suffices to create the base. The reason is that the creation of edges in the bottom-up rule depends only on the existence of a passive edge. In a parser using the top-down rule, however, we must also create empty active edges from the first symbol of the string being parsed to itself labeled with productions having the start symbol of the grammar as left-hand side. This is because such a rule uses an *active* edge to create another one.

Agenda. The rules for producing edges that we have described only *create* edges, but do not *add* them to the chart. Normally, chart parsers store edges in two different data structures: the chart and an *agenda* of the set of edges to be added to it. The choice of the procedure for selecting edges from the agenda to be added to the chart is a degree of freedom relegated to the chart-parser designers. When an edge is removed from the agenda, it is added to the chart only if it has not been added before.

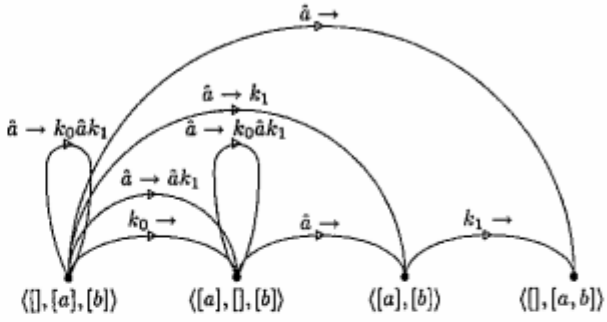

Figure 4: *A chart constructed with the top-down rule.*

Example. Figure 4 shows a chart created by a parser using the top-down rule for the grammar with productions:

$$\hat{a} \to k_0\ \hat{a}\ k_1$$
$$\hat{a} \to \langle [a], [], [b] \rangle$$
$$k_0 \to \langle [], [a], [b] \rangle$$
$$k_1 \to \langle [a], [b] \rangle$$

and the input string $\langle [], [a], [b] \rangle$ $\langle [a], [], [b] \rangle$ $\langle [a], [b] \rangle$ $\langle [], [a, b] \rangle$. Terminals have been enclosed in angled brackets. The last symbol $\langle [], [a, b] \rangle$ is not part of the string itself, but rather an end marker. This example will be used again to illustrate the chart created by a proof procedure when concatenating $[a]$ to $[b]$.

Phillips' variant of the bottom-up parser. Phillips observed [Simpkins and Hancox 1990] that the bottom-up chart parser can be modified so that some edges can be disposed of as the chart is built. The agenda, then, only keeps passive edges, *ordered with respect to the position of the symbol on the string they start from.* The chart only keeps active edges. When the first passive edge E is removed from the agenda and momentarily added to the chart, then

1. the fundamental rule is applied as many times as possible, and

2. the bottom-up rule is also applied if possible, followed by applications of the fundamental rule.

In both cases, if the resulting edges are active, they are added to the chart; otherwise they are added to the agenda. After this, E can be disposed of. The reason is that E cannot contribute to the creation of any more new edges.

3 Chart parsers as proof procedures

In this section we will show that chart parsers can be regarded as proof procedures for compositional programs.

State-oriented computations. The difference-list representation of strings associates a production

$$P_0 \rightarrow P_1 \cdots P_n \tag{1}$$

with a clause of the form

$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1), \ldots, p_n(X_{n-1}, X_n) \tag{2}$$

and a production with a single terminal on its right-hand side

$$P \rightarrow a \tag{3}$$

with

$$p([a|X], X) \leftarrow \tag{4}$$

With a programming language having only those clauses we cannot compute all computable functions. But if we generalize (4) to

$$p(t, t') \leftarrow \tag{5}$$

where t and t' are terms such that $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, we can. (Throughout, $\mathrm{var}(t)$ denotes the set of variables occurring in term t.) This can be shown, for instance, by associating a logic program with a flowchart in such a way that both have the same set of computations [Clark and van Emden 1981]. A refutation for such a program and a query with a ground term in its first argument may be said to define a sequence of ground terms, resembling the sequence of states in a computation of a programming language using destructive assignment. Thus we shall say that such a logic program defines *state-oriented computations.*

Strings vs. state-oriented computations. There are two main differences between state-oriented computations and strings. One is that at a given point of a state-oriented computation, there may be more than one way to extend it. State-oriented computations are then said to be nondeterministic. This phenomenon does not occur in strings, which have a linear structure.

The other difference is that whereas we do know all the symbols of the string before it is parsed, we do not know initially all the states in a computation. A proof procedure could in principle compute some sequence of states before trying to build a chart. However, it may not be convenient to do so, because not all sequences of states form the base of a chart. A better idea is to extend the computations one step at a time, guided by the part of the chart built so far.

Chart parsers as proof procedures. We shall generalize chart parsers to proof procedures by establishing a correspondence between chart parsing and resolution.

The difference-list representation of languages suggests that clauses of the form (2) should play the role of productions with no terminals on the right-hand side (1). Clauses of the form (5) would then be the counterpart of productions with exactly one terminal on the right-hand side (3).

Given this correspondence, we now turn our attention to edges. The fundamental rule of chart parsing takes two edges and produces another one. Resolution, on the other hand, takes two clauses and produces another one. This suggests identifying edges with clauses and the fundamental rule with a resolution step.

The fundamental rule. If an edge from **a** to **b** labeled with $P_0 \rightarrow P_i \cdots P_n$ corresponds to a clause of the form

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n) \tag{6}$$

then the fundamental rule corresponds to a resolution step having (6) (which plays the role of the active edge) and

$$p_i(\mathbf{b}, \mathbf{c}) \leftarrow$$

(which plays the role of the passive edge) as input clauses. The resolving clause of this resolution step is

$$p_0(\mathbf{a}, X_n) \leftarrow p_{i+1}(\mathbf{c}, X_{i+1}), \ldots, p_n(X_{n-1}, X_n)$$

which corresponds to an edge from **a** to **c** labeled with $P_0 \rightarrow P_{i+1} \cdots P_n$. By correctness of resolution, the resolving clause is a logical consequence of the two input clauses. Thus, we have generalized the fundamental rule to a correct operation.

The top-down and the bottom-up rules. Given the above identification of clauses with edges, the top-down rule for parsing corresponds to the following. Let P be a program in compositional form. If a clause of the form

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n)$$

is added to the chart, then create a clause of the form

$$p_i(\mathbf{b}, X_m) \leftarrow q_1(\mathbf{b}, X_1), \ldots, q_m(X_{m-1}, X_m)$$

for every clause in P of the form

$$p_i(X_0, X_m) \leftarrow q_1(X_0, X_1), \ldots, q_m(X_{m-1}, X_m)$$

The created clause is an instance of a clause in P, which is a logical consequence of P. The bottom-up rule can be generalized in a similar way.

The base. The base can be extended one step at a time as follows. For each clause

$$p_0(\mathbf{a}, X_n) \leftarrow p_i(\mathbf{b}, X_i), \ldots, p_n(X_{n-1}, X_n)$$

that is created, create a clause

$$r(\mathbf{b}, t'\theta) \leftarrow$$

for each clause in P of the form

$$r(t, t') \leftarrow$$

such that $\mathbf{b}$ and t unify with unifier θ and there is a path from p_i to r. There is a *path* from p to r if

1. p is r or
2. there is a clause in P of the form

$$p(X_0, X_m) \leftarrow q(X_0, X_1), \ldots, s(X_{m-1}, X_m)$$

and there is a path from q to r.

4 Conversion of fixed-mode to compositional programs

We have seen that chart parsers can be regarded as proof procedures for compositional programs. However, logic programs are not normally written in compositional form. In this section we observe that it is possible to convert a fixed-mode logic program with ground data structures into compositional form. The resulting program is logically implied by an extension of the original one.

First we define the class of programs transformable by our method and the class produced by it. Then we prove, for a particular example, the correctness of the resulting program. We omit the proof for the general case, which can be found in [Rosenblueth 1991].

4.1 Directed and compositional programs

Directed form. The class of transformable programs has fixed modes. Thus we assume, without 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. A definite clause of the form

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

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. each variable occurring in t'_i occurs only once in t'_i, for $i = 0, \ldots, n$

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 query 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.

Compositional form. A *compositional clause* is a definite clause of the form

$$p(t, t') \leftarrow \quad \text{or}$$
$$p_0(X_0, X_n) \leftarrow p_1(X_0, X_1), \ldots, p_n(X_{n-1}, X_n) \quad (n > 0)$$

where t and t' are terms such that $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, and the X_i are distinct variables. A logic program with only compositional clauses is a *compositional program.*

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 query yields a *conditional answer* [Vasey 1986]. Such an answer is a clause in which the head is the subgoal in the initial query of that derivation with the composition of the substitutions applied to it, and the body is the set of subgoals in the last query of that derivation.

4.2 Example

We illustrate our method with the following program for concatenating two lists. It defines the usual *append* relation, but its arguments have been grouped in such a way that its two inputs constitute the first argument, and its output, the second. $a(\langle X, Y\rangle, \langle Z\rangle)$ holds if Z is the concatenation of the list Y at the end of the list X. 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.

$$a(\langle [\,], Y\rangle, \langle Y\rangle) \leftarrow \tag{7}$$
$$a(\underbrace{\langle [W|X], Y\rangle}_{t_0}, \underbrace{\langle [W|Z]\rangle}_{t'_1}) \leftarrow a(\underbrace{\langle X, Y\rangle}_{t'_0}, \underbrace{\langle Z\rangle}_{t_1}) \tag{8}$$

We shall convert (8), which is directed, to compositional form. This process can be motivated as follows.

Assume that we wish to construct an SLD-derivation for (7) and (8) with a query having a ground input that unifies with the head of (8). It is necessary, then, to remember the term with which W unifies, to be able to add it to the front of the result of appending the lists that unify with X and Y. This lack of information in the arguments of the subgoal of (8) prevents us from representing a computation by the composition of the relation denoted by $a(X,Y)$ with itself. To be able to use relational composition for representing computations, we must provide the missing information to the arguments. A common technique in the implementation of state-oriented languages for recording values needed in subsequent steps of a computation is the use of a stack. This suggests storing the term unifying with W in a list that is treated as a stack. We thus define the predicate:

$$\hat{a}(\langle St_0|X\rangle, \langle St_1|Y\rangle) \leftrightarrow St_0 = St_1 \;\&\; a(X,Y) \qquad (9)$$

Although both St_0 and St_1 represent the same stack, it will be convenient to keep two 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 of the *standard equality theory*. This theory consists of the following axioms:

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

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

Next, we can derive another clause in which the input and the output of the atoms have no variables in common:

$$a(\langle [W|X], Y\rangle, \langle [W'|Z]\rangle) \leftarrow W = W', a(\langle X, Y\rangle, \langle Z\rangle) \qquad (10)$$

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

Next we can proceed as follows. Unfolding[1] (10) on the "if" part of the definition of $\hat{a}$ (9) we obtain:

$$\hat{a}(\langle St_0, [W_0|X_0], Y_0\rangle, \langle St_1, [W_1|Z_1]\rangle) \leftarrow St_0 = St_1, W_0 = W_1, a(\langle X_0, Y_0\rangle, \langle Z_1\rangle)$$

Next 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:

$$\hat{a}(\langle St_0, [W_0|X_0], Y_0\rangle, \langle St_1, [W_1|Z_1]\rangle) \leftarrow [W_0|St_0] = [W_1|St_1], a(\langle X_0, Y_0\rangle, \langle Z_1\rangle)$$

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

$$\hat{a}(\langle St_0, [W_0|X_0], Y_0\rangle, \langle St_1, [W_1|Z_1]\rangle) \leftarrow \hat{a}(\langle [W_0|St_0], X_0, Y_0\rangle, \langle [W_1|St_1], Z_1\rangle) \qquad (11)$$

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 with another name (W_1) and finally added to the front of the result of appending the tail of the first list to the second.

The fact that in (11) the inputs share no variables with the outputs allows us to fold the definitions of k_0 and k_1:

$$\begin{aligned}
k_0(U,V) \leftrightarrow \exists St_0 \exists W_0 \exists X_0 \exists Y_0.[&\langle St_0, [W_0|X_0], Y_0\rangle = U \\
&\;\&\; \langle [W_0|St_0], X_0, Y_0\rangle = V] \\
k_1(U,V) \leftrightarrow \exists St_1 \exists W_1 \exists Z_1.[&\langle [W_1|St_1], Z_1\rangle = U \\
&\;\&\; \langle St_1, [W_1|Z_1]\rangle = V]
\end{aligned}$$

in the following clause:

$$\begin{aligned}
\hat{a}(U_0,U_3) \leftarrow\; &\langle St_0, [W_0|X_0], Y_0\rangle = U_0, \\
&\langle [W_0|St_0], X_0, Y_0\rangle = U_1, \\
&\langle [W_1|St_1], Z_1\rangle = U_2, \\
&\langle St_1, [W_1|Z_1]\rangle = U_3, \\
&\hat{a}(U_1,U_2)
\end{aligned}$$

which is a logical consequence of (11) 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 [Shoenfield 1967 p. 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 compositional form of a directed program may be obtained in a more straightforward manner based on the theorem in the Appendix.

4.3 Example (continued)

The compositional form of the *append* program used to concatenate lists is, then:

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

[1] 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.

The chart created by a proof procedure using the top-down rule for this program and the query $\leftarrow \hat{a}(\langle [], [a], [b] \rangle, Z)$ was shown in Figure 4.

5 A comparison with Pereira and Warren's Earley deduction

Pereira and Warren [1983] have extended Earley's [1970] algorithm to a proof procedure for logic programs that 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 [Garey and Johnson 1979].

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 than 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

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

With subsumption, Earley deduction does terminate for this example. Our method, in contrast, does not require subsumption and yet also terminates.

We have implemented Earley deduction based on the top-down chart parser of [Gazdar and Mellish 1989, p. 211, 212]. and using Robinson's [1965] subsumption algorithm as modified in [Gottlob and Leitsch 1985]. We have also adapted both top-down and bottom-up parsers [Gazdar and Mellish 1989, p. 208–212] to proof procedures for compositional programs. In addition, we have modified Phillips' variant of the bottom-up chart parser as presented in [Simpkins and Hancox 1990]. The following table summarizes execution times for several programs and queries. The tests were performed on a SUN SPARC station 1 using SICStus Prolog.

	PW1	top-down		Phillips		PW2	
	time	time	su	time	su	time	su
perm	48	46	1.0	11	4.4	7	6.9
hanoi	36	21	1.7	9	4.0	2	18.0
append	49	22	2.2	5	9.8	6	8.2
qsort	249	30	8.3	7	35.6	17	14.6

"perm" computes all permutations (four elements), "hanoi" solves the Towers of Hanoi problem using difference lists to store the sequence of steps of the solution (five disks), "append" is the ordinary append used to concatenate lists (80 elements), and "qsort" is quicksort using difference lists (20 elements). "PW1" is Pereira and Warren's proof procedure, "top-down" and "Phillips" result from our method, and "PW2" is a variant of Pereira and Warren's proof procedure in which subsumption has been replaced by a syntactic equality test. "su" stands for "speedup." Times are in seconds.

6 Concluding remarks

Chart parsers work for a generalization of the difference-list representation of context-free grammars. This generalization replaces the clauses representing productions with exactly one terminal by clauses having terms subject to only one syntactic restriction: all variables in the second argument must appear in the first (compositional programs).

It is possible to transform [Rosenblueth 1991] fixed-mode logic programs into this generalization by adding arguments that play the role of a stack. Consequently, chart parsers can be used as proof procedures for fixed-mode logic programs transformed by this method. Strings correspond to sequences of ground terms.

Experiments have shown that programs so transformed can be executed several times faster than with the previous adaptation of Earley's parser to a proof procedure done by Pereira and Warren [1983].

Phillips has modified [Simpkins and Hancox 1990] the bottom-up chart parser so that portions of the chart being built can be disposed of. It is essential in the doctored parser to keep edges ordered with respect to the string

being parsed. In compositional programs, computations form sequences and Phillips' idea can also be applied. It is not clear how to apply it to Pereira and Warren's method.

Proof procedures obtained from chart parsers terminate for some programs for which Prolog does not. In addition, it is possible to build charts in parallel [Trehan and Wilk 1988].

Acknowledgments

We are grateful to Felipe Bracho, Carlos Brody, Warren Greiff, Rafael Ramírez, Paul Strooper, and Carlos Velarde. The anonymous referees also made valuable suggestions. We acknowledge the facilities provided by IIMAS, UNAM.

Bibliography

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

[Earley 1970] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 14:453–460, 1970.

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

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

[Gottlob and Leitsch 1985] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the ACM*, 32(2):280–295, 1985.

[Pereira and Warren 1983] Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. Technical Report 295, SRI, June 1983.

[Robinson 1965] J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

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

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

[Simpkins and Hancox 1990] Neil K. Simpkins and Peter Hancox. Chart parsing in Prolog. *New Generation Computing*, 8:113–138, 1990.

[Trehan and Wilk 1988] R. Trehan and P.F. Wilk. A parallel chart parser for the committed choice nondeterministic logic languages. In K.A. Bowen and R.A. Kowalski, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 212–232. MIT Press, 1988.

[Vasey 1986] P. 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.

Appendix

Our method for converting fixed-mode programs to compositional form is based on the following theorem, which is proved in [Rosenblueth 1991].

Theorem 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

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

for $i = 1, \ldots, n$. Then the clause

$$\begin{aligned}\hat{p}_0(X_0, X_{2n+1}) \leftarrow\ & k_0(X_0, X_1), \hat{p}_1(X_1, X_2),\\ & k_1(X_2, X_3), \hat{p}_2(X_3, X_4), \ldots,\\ & \hat{p}_n(X_{2n-1}, X_{2n}), k_n(X_{2n}, X_{2n+1})\end{aligned}$$

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:

$$\hat{p}_i(\langle St|X\rangle, \langle St'|Y\rangle) \leftrightarrow St = St' \ \&\ p_i(X, Y) \qquad i = 0, \ldots, n$$

$$k_0(U, V) \leftrightarrow \exists Y_{1,0} \cdots \exists Y_{m_0,0}.[\langle St|t_0\rangle = U \ \&\ \langle \Sigma_1|t'_0\rangle = V]$$

$$k_1(U, V) \leftrightarrow \exists Y_{1,1} \cdots \exists Y_{m_1,1}.[\langle \Sigma_1|t_1\rangle = U \ \&\ \langle \Sigma_2|t'_1\rangle = V]$$

$$\vdots$$

$$k_{n-1}(U, V) \leftrightarrow \exists Y_{1,n-1} \cdots \exists Y_{m_{n-1},n-1}.[\langle \Sigma_{n-1}|t_{n-1}\rangle = U \ \&\ \langle \Sigma_n|t'_{n-1}\rangle = V]$$

$$k_n(U, V) \leftrightarrow \exists Y_{1,n} \cdots \exists Y_{m_n,n}.[\langle \Sigma_n|t_n\rangle = U \ \&\ \langle St|t'_n\rangle = V]$$

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 $[X_{1,i}, \ldots, X_{d_i,i}|St]$, and $\{X_{1,i}, \ldots, X_{d_i,i}\} = \Pi_i$, for $i = 1, \ldots, n$.