# Earley Deduction[1]

*Harry H. Porter III*
*Portland State University*

March 10, 1986

Author's e-mail: harry@cs.pdx.edu
Author's Web Page: www.cs.pdx.edu/~harry

This paper is online at:
www.cs.pdx.edu/~harry/earley/earley.pdf
www.cs.pdx.edu/~harry/earley/earley.htm

## Abstract

This paper first reviews Earley Deduction, a generalization of the Earley Parsing Algorithm to the execution of Horn Clause Logic Programs. Earley Deduction is both sound and—unlike the standard Prolog interpreter—complete; proofs of this are included. For functor-free programs, the method is also guaranteed to terminate. (The functor-free subset of Prolog is called DATALOG and can be used to compute relational join, selection, transitive closure, etc.) Earley Deduction is elegant but, unfortunately, the price paid for completeness is efficiency. We conclude by describing implementation techniques that make Earley Deduction practical for DATALOG programs.

## Review of Earley Deduction

The traditional Prolog interpreter performs a depth-first search for a proof and consequently suffers from some of the problems associated with depth-first parsing algorithms. For example, if a program contains left-recursive rules the interpreter may go into an infinite loop, failing to find a proof even though one exists, just as a top-down parser may fail to find a parse when given a grammar containing left-recursion. In 1970, Jay Earley described an algorithm called *Earley Parsing* that overcomes some of the problems of both top-down and bottom-up parsing by combining aspects of both [Earley, 1970]. In computational linguistics, various incarnations of this algorithm have been studied under the name *Chart Parsing*.

[1] Although written long ago, when I was a Ph.D. student at Oregon Graduate Center / Oregon Graduate Institute / OHSU, this paper was reformatted in May, 2009, and posted to the web at this time. Other than repairing typos, no substantive changes have been made.

F. C. N. Pereira and D. H. D. Warren have extended this algorithm to the execution of logic programs and call the method *Earley Deduction* [Pereira & Warren, 1983]. Their algorithm has three desirable properties. First, it is sound and complete. Second, it is guaranteed to terminate for an interesting subset of logic, namely functor-free programs. Finally, Earley Deduction is straightforward and easy to implement.

We begin by describing Earley Deduction by tracing its execution on a small logic program. Then we show correctness and completeness of the algorithm and show that it terminates for functor-free programs. We conclude with a discussion of implementation techniques and results for a version of the algorithm restricted to functor-free programs.

The following example logic program contains a transitive closure rule which causes nontermination in the standard interpreter. The clauses comprising our example are:

```
p(X,Z) ← p(X,Y), p(Y,Z).        (1)
p(a,b).                         (2)
p(b,c).                         (3)
```

These are called the *program clauses*.

A goal is transformed into a *goal clause* by adding a dummy predicate `ans` as the clause head. The arguments to this `ans` predicate will be used to accumulate the bindings computed in a successful proof. The goal clause we will use is:

```
ans(Z) ← p(a,Z).                (4)
```

This goal has only one literal but, in general, there will be several literals on the right-hand side.

The method works by building up a set of *derived clauses*. As an initialization step, the *goal clause* is added as the first element to the set of derived clauses. Each step of the method adds another clause to the set of derived clauses and, when no more clauses can be added, terminates.

There are two inference rules called *reduction* and *instantiation*. Both rules work by combining a derived clause with another clause (either program or derived). The former (derived) clause will be called the *selected clause* and the latter clause will be called the *other clause*. One literal within the body of each derived clause will be marked as the *selected literal*. It is chosen when the clause is first created and added to the derived clause set. The choice is arbitrary; we will always select the first literal in the body of a derived clause.

The reduction step can only be applied when the *other* clause is a unit clause. It works as follows. First, the selected literal of the selected clause is unified with the unit clause. The selected clause must always be a derived clause but the other clause—the unit clause—can be either a program or a derived clause. Let $\sigma$ be the most general unifier. [Just as in Prolog, variables are assumed to be quantified over individual clauses: variables in different clauses are assumed to be distinct, even if they share the same name.]

For example, let clause (4) be the selected clause and let clause (2) be the other clause. The selected literal of the selected clause is `p(a,Z)` since it is the left-most literal on the right-hand side. The unifier is σ = { `Z` ← `b` }.

Second, remove the selected literal from the selected clause, apply σ to what remains and add the result as a new derived clause. Removing the selected literal gives `ans(Z)` and applying the unifier gives `ans(b)` which is added to the derived clause set.

```
ans(b).                                                    (5)
```

Whenever a clause is derived that has head predicate `ans` and that is a unit clause, it is output as a solution. Thus, `ans(b)` is printed as an answer.

The second kind of rule is *instantiation*. For this rule, we take the selected literal of the selected clause and unify it with the positive literal (i.e., the head) of a non-unit program clause, giving a most general unifier σ. We then apply σ to the program clause and add the result as a new derived clause.

To illustrate this rule, we use clause (4) as the selected clause to instantiate clause (1). The unification of the selected literal `p(a,Z)` with the head of clause (1) `p(X,Z)` gives σ = { X ← a }. Instantiating the program clause with σ gives the new derived clause.

```
p(a,Z) ← p(a,Y), p(Y,Z).                                   (6)
```

Continuing the inferencing, the reduction rule can be applied to clause (2) and clause (6). We say that clause (2) *reduces* clause (6) and the result is clause (7):

```
p(a,Z) ← p(b,Z).                                           (7)
```

It is occasionally possible to perform a reduction or instantiation step producing a clause that has already been derived earlier. For example, clause (6) can now be used to instantiate clause (1) but the result has already been derived (as clause (6) itself). To avoid this redundancy, we stipulate that a clause is not to be added as a new derived clause if it is subsumed by an already-derived clause. (A more general term *subsumes* a more specific term if the latter can be obtained by applying a substitution to the former.) The obvious way to perform this check is to take a new candidate clause and look through all the derived clauses, performing the subsumption check on each. This blind searching can be quite time consuming and we will have something to say below about doing it more intelligently.

We complete the specification of Earley Deduction by specifying how the algorithm selects pairs of clauses for combination. Not every selection strategy will find answers even when they exist, as the following sequence demonstrates:

# Earley Deduction

**Program Clauses:**

| | | |
|---|---|---|
| `p(X) ← p(f(X)).` | | (14) |
| `p(a).` | | (15) |

**Derived Clauses:**

| | | |
|---|---|---|
| `ans ← p(a).` | Goal | (16) |
| `p(a) ← p(f(a)).` | 16 instantiates 14 | (17) |
| `p(f(a)) ← p(f(f(a))).` | 17 instantiates 14 | (18) |
| `p(f(f(a))) ← p(f(f(f(a)))).` | 18 instantiates 14 | (19) |

•
•
•

At some step in the algorithm let clauses $\mathbf{C}_1$ and $\mathbf{C}_2$ be two clauses that can be combined (either using instantiation or reduction) to produce a new clause $\mathbf{C}_3$. We require the selection strategy to eventually get around to combining $\mathbf{C}_1$ and $\mathbf{C}_2$ and considering $\mathbf{C}_3$. Consequently, a clause at least as general as $\mathbf{C}_3$ will eventually be added to the derived set, since $\mathbf{C}_3$ itself will be added unless it is subsumed by some other previously derived clause. We call such a selection strategy *fair*.

Assume the program clauses are numbered from 1 to n, clause n+1 is the goal clause and new derived clauses are numbered sequentially as they are added from n+2. Here is an example of a fair scheduling policy:

```
i := n+1
repeat
   for j := 1 to i-1 do
      Attempt to combine clause i and clause j using instantiation and
      reduction and add any new clauses to the set of derived clauses.
   endfor
   i := i+1
until i > the number of clauses
```

There are several more instantiations and reductions we can perform before we reach a point where no new clauses can be derived. This example program quickly terminates and we show the resulting clauses below. We have included comments and, for convenience, clauses (1) through (7) are repeated.

# Earley Deduction

**Program Clauses:**

| | | |
|---|---|---|
| `p(X,Z) ← p(X,Y), p(Y,Z).` | | (1) |
| `p(a,b).` | | (2) |
| `p(b,c).` | | (3) |

**Derived Clauses:**

| | | |
|---|---|---|
| `ans(Z) ← p(a,Z).` | Goal | (4) |
| `ans(b).` | 2 reduces 4 | (5) |
| `p(a,Z) ← p(a,Y), p(Y,Z).` | 4 instantiates 1 | (6) |
| `p(a,Z) ← p(b,Z).` | 2 reduces 6 | (7) |
| `p(b,Z) ← p(b,Y), p(Y,Z).` | 7 instantiates 1 | (8) |
| `p(a,c).` | 3 reduces 7 | (9) |
| `p(b,Z) ← p(c,Z).` | 3 reduces 8 | (10) |
| `p(c,Z) ← p(c,Y), p(Y,Z).` | 10 instantiates 1 | (11) |
| `ans(c).` | 9 reduces 4 | (12) |
| `p(a,Z) ← p(c,Z).` | 9 reduces 6 | (13) |

In examining these clauses, note how the right-hand sides of derived non-unit clauses represent goals we have discovered that need solving in order to produce an answer. For example, the right-hand side of clause (7) indicates that we need to solve `p(b,Z)` in order to complete a proof. The head of clause (7) is the subgoal that motivates the proof of `p(b,Z)`, namely `p(a,Z)`. The goal `p(b,Z)` was encountered while trying to solve the body of clause (6) and was produced when clause (2) reduced clause (6).

The algorithm has a top-down component since new goals are only produced when needed to satisfy existing goals. Although this example is too short to illustrate it, goals are only created when their solution is relevant in solving the goal clause. The algorithm also has a strong bottom-up flavor since once proven, solutions are stored and reused, rather than recomputed. For example, once a solution for `p(a,Z)` is produced (e.g. clause (9)) it can be used by both clause (4) and clause (6).

## Soundness of Earley Deduction

We next give an informal argument that this proof procedure is correct (*sound*) in the sense that any answer obtained implies the query is a logical consequence of the axioms. We assume the reader is familiar with refutation proofs [Robinson, 1965]. We first show that both inference rules are sound. Then, by induction on the sequential numbering of the derived clauses, each derived clause is a logical consequence of the program clauses and goal clause.

Recall that a *definite clause* is a disjunction of literals, one positive and several negative, although it is more intuitively written using an implication whose antecedent is a conjunction of positive literals. The query, a conjunction of positive literals, is negated (and then re-written as a disjunction of negative literals) and the proof is by refutation. The contradiction is represented by the *empty clause*. In the Earley Deduction Algorithm, the unit clause `ans(...)` denotes the empty clause and also carries information about the binding obtained in its derivation. This technique of adding additional dummy literals to clauses to accumulate answer substitutions is well-known (e.g. [Nilsson, 1971]).

In a traditional refutation proof, there is only one inference rule: resolution. Here we have 2 rules, instantiation and reduction. Reduction is clearly a special case of resolution, namely when one of the two clauses consists of a single positive literal. A clause produced by instantiation can also be seen to be a logical consequence of previous clauses since it is just an instantiated version of an axiom. Thus, if the unit clause `ans(...)` is derived, the empty clause has been produced and thus the refutation is complete. Figure 1, which shows graphically the relationships between derived clauses in the proof of `ans(c)`, may make the correspondence between Earley Deduction proofs and the resolution proof process clearer.

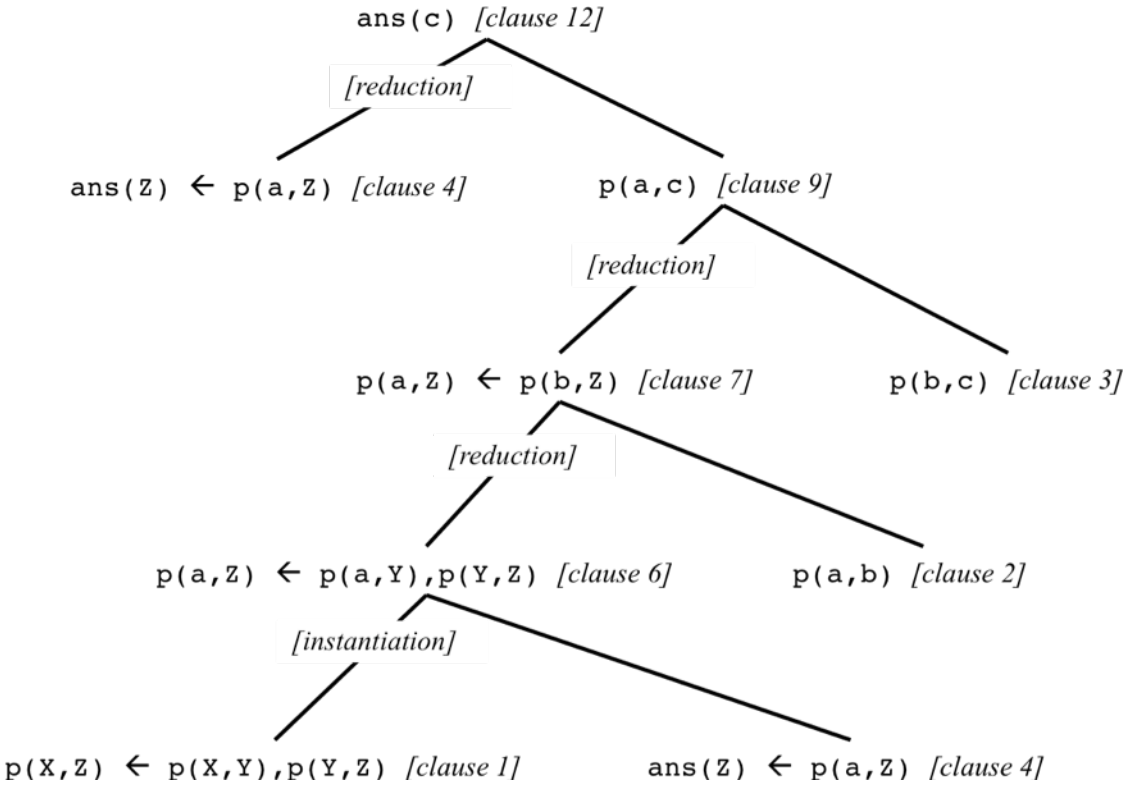


**Figure 1.** An Earley Deduction Tree.

## Completeness of Earley Deduction

We will call trees like the one in Figure 1 *Earley Deduction Trees*.[2] The tree will always have the empty clause `ans(...)` at the root and every node will either (1) have two children or (2) be a program clause, goal clause or derived unit clause. If the node has two children, it will have been produced from those clauses by either reduction or instantiation. For clauses produced by reduction, one child will be a unit clause, since one of the clauses used in the reduction step must

[2] Comments are shown in brackets and, technically, are not part of the tree.

be a unit clause. Clauses produced by instantiation will simply be less general instantiations of some program rule.

In showing that Earley Deduction is complete, we assume that the query is provable (thus a proof exists) and show that the algorithm will find a proof, expressed as an Earley Deduction Tree. If the query is true, then a Prolog proof tree (to be described below) exists, although the Prolog interpreter will not necessarily find it. We show how this tree can be converted into an Earley Deduction Tree and then show that Earley Deduction will discover a tree at least as general.

Figure 2 is an example Prolog proof tree. The root node is the query clause and every other node is an instantiated clause from the program. Note that the substitutions (which some authors just attach to the clauses) have already been performed on the clauses. In the example, the substitutions happened to have eliminated all variables but that will not always be the case. Also note that the head of a child clause exactly matches one negative literal in its parent's right-hand side and that each negative literal in a parent clause matches the positive literal of one of its children. If a refutation proof exists, then such a tree exists.

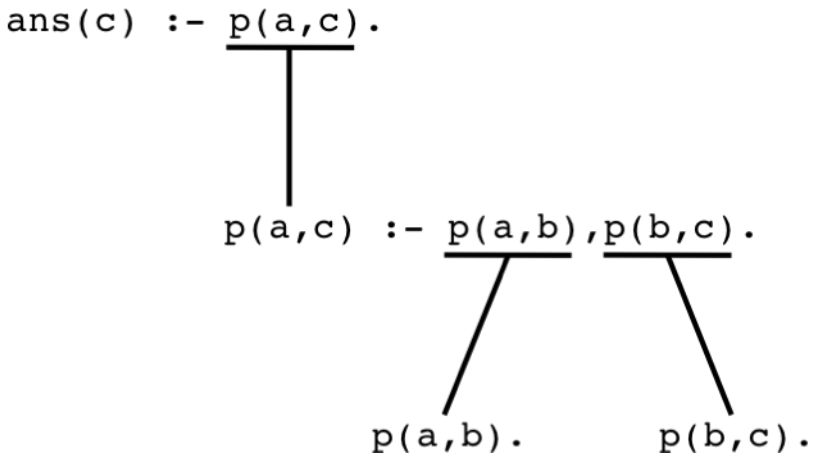


**Figure 2.** A Prolog Proof Tree.

Next, we show how to construct an Earley Deduction Tree from a Prolog proof tree. Call the result the *constructed tree*. In the construction, we will associate an Earley tree with every *node* in the Prolog tree. The construction is defined recursively, starting at the leaves of the Prolog tree and working toward the root. The Earley tree associated with the root is the result.

The leaves of the Prolog tree are unit clauses from the program. With these, associate one-node Earley trees consisting of the same program unit clauses.

The translation of interior nodes of the Prolog trees is shown diagrammatically in Figures 3 and 4. Figure 3 shows an interior node of the Prolog tree and the roots of the Earley trees associated with each of its children. Figure 4 shows how to build the Earley tree to be associated with the interior node of the Prolog tree shown in Figure 3.

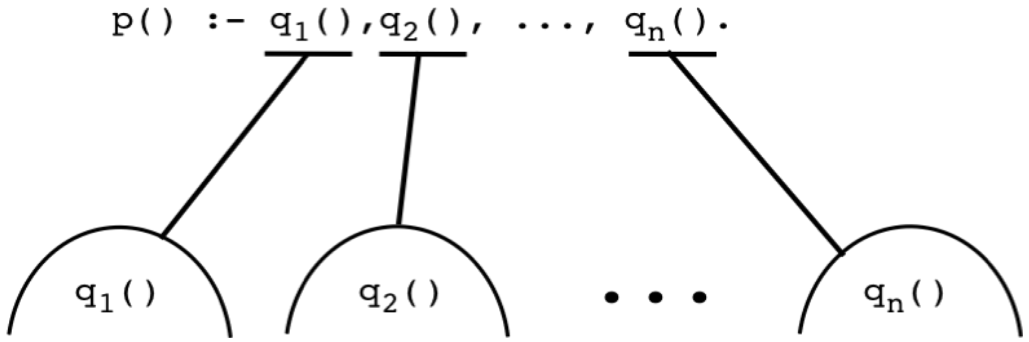


**Figure 3.** An Interior Node.

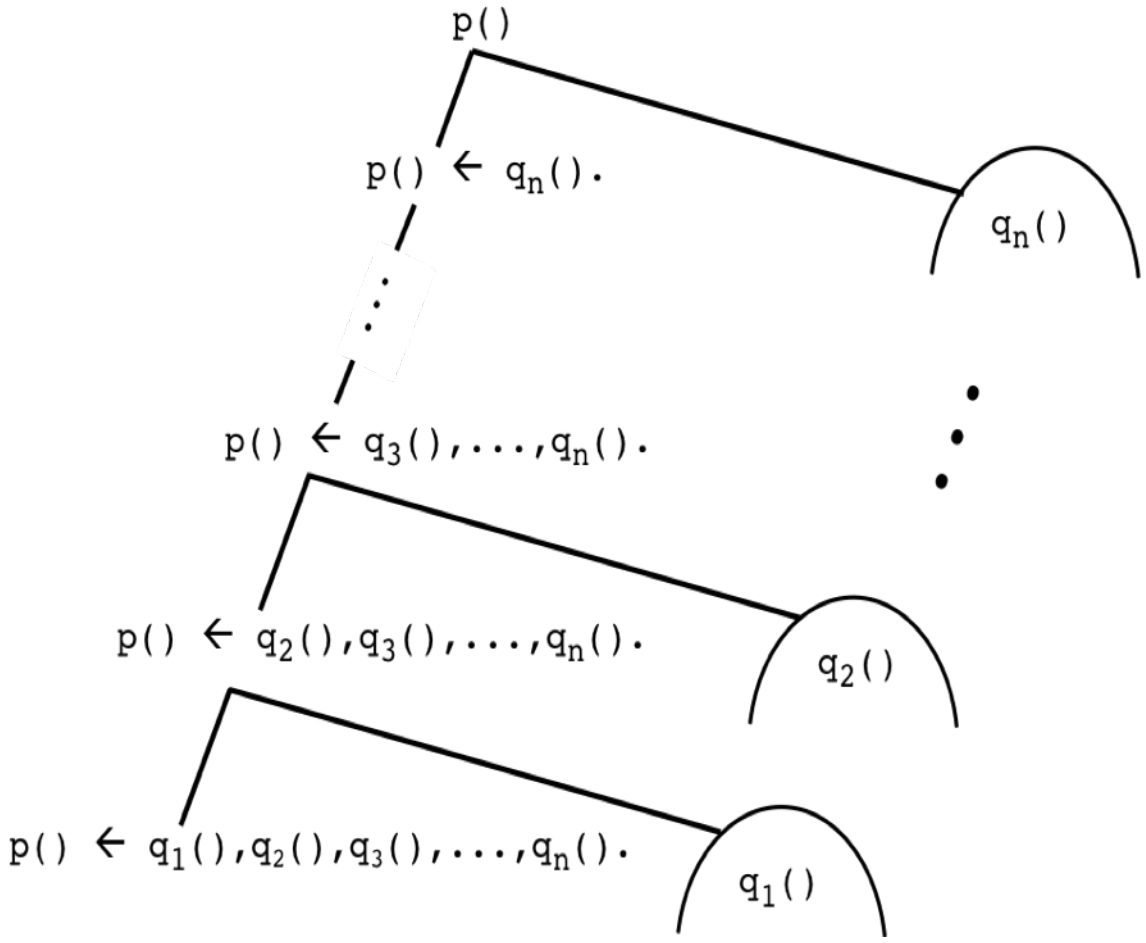


**Figure 4.** Resulting Earley Deduction Tree.

As an example, Figure 5 shows the constructed tree associated with the Prolog proof tree shown in Figure 2.

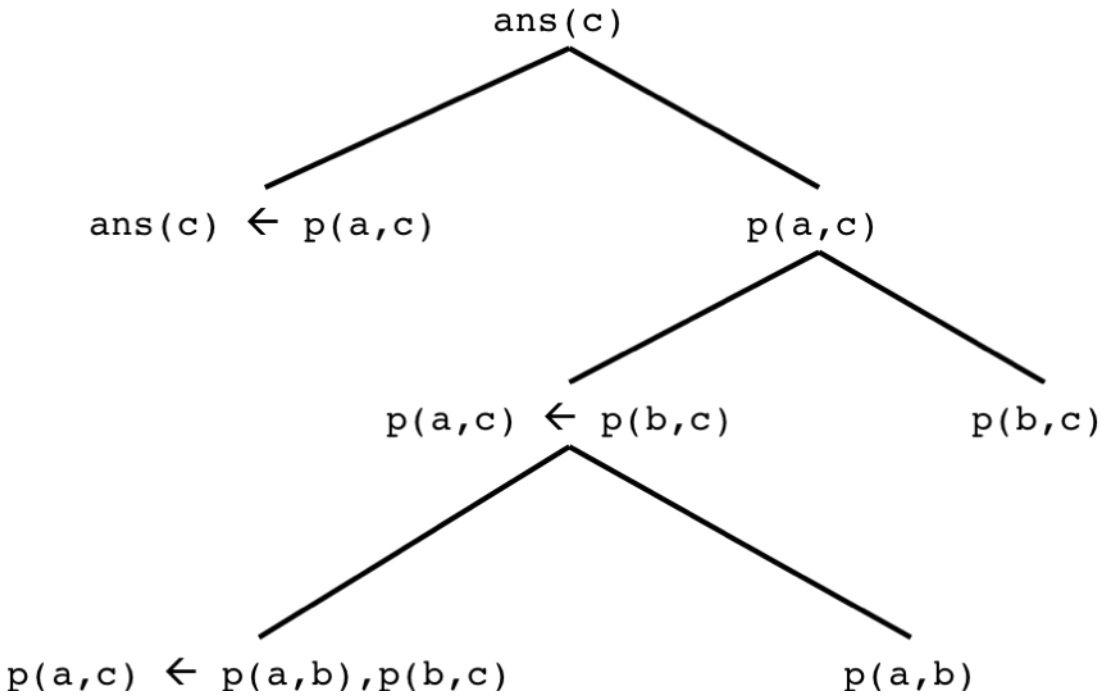


**Figure 5.** An Example Earley Deduction Tree.

Next, consider an interior node of the constructed Earley tree built from the Prolog proof tree. It is labeled with a clause $\mathbf{C}_3$ and has two children whose roots are labeled with clauses $\mathbf{C}_1$ and $\mathbf{C}_2$. If the Earley Deduction algorithm has already produced two clauses that are at least as general as $\mathbf{C}_1$ and $\mathbf{C}_2$ then, because the selection strategy is fair, they will ultimately be combined to derive a clause at least as general as $\mathbf{C}_3$.

Finally, note that the Earley Deduction algorithm begins with a collection of program and goal clauses containing clauses at least as general as the clauses labeling the leaves of the constructed tree. By induction on the size of the constructed tree, we conclude that the algorithm will ultimately derive a clause which is at least as general as the head of the goal clause labeling the root of the Prolog proof tree. Thus, Earley Deduction is complete and will eventually find any existing proof.

## Termination for DATALOG

A DATALOG program is just a definite clause program that contains no functors. Our example was such a program. The functor-free subset of definite clause logic is important because it can be used to express data and queries from relational algebra in a clear and concise way. In addition, it can be used to express queries involving recursively defined data (e.g. transitive closure queries) which are problematic in the relational paradigm.

Prolog is not altogether satisfactory for executing such queries because left-recursive rules may cause non-termination and avoiding left-recursive rules may be inconvenient. Fortunately, Earley Deduction is guaranteed to terminate whenever the program clauses contain no functors. We next restrict our attention to DATALOG programs and show this.

Every step of the deduction adds a new clause to the set of derived clauses but these clauses are never any longer than the longest program clause.[3] To see that infinitely long clauses can never be derived, consider the reduction and instantiation rules. Reduction takes a given clause (the selected clause) and removes the selected literal. Thus reduction can't be used to make longer clauses. Instantiation takes a program clause and instantiates it and so it can't be used to make a longer clause either.

We assume that the DATALOG program has a finite number of clauses, each with a finite number of literals and each of these with a finite number of arguments. Thus, there are a finite number of predicates and constant symbols appearing in the program and goal clauses. Given a finite number of symbols and some size k, there are only a finite number of clauses of length k or fewer symbols. (Two clauses differing only in the names of variables are considered equal, so an infinite supply of variable names does not effect this bound.)

The only question left is: Can we get stuck infinitely looking for but never finding a new derived clause? At any moment, there are only a finite number of pairs of derived and program clauses and there are only a couple of ways to combine each pair to produce a new clause. Given a newly produced clause, the subsumption check can also be done in finite time. So, if there is another clause that can be derived, the procedure must eventually find it.

Thus, since we are guaranteed to ultimately produce any clause that can be derived (by completeness) and since we are guaranteed to never produce clauses longer than a given bound and since there are only a finite number of such clauses, the process is guaranteed to terminate. Of course, when we allow functor symbols, the derived clauses may be longer than either of the two existing clauses, so the procedure is not guaranteed to terminate for logic programs in general.

## Implementation

We have implemented Earley Deduction using Smalltalk on a Tektronix 4400 personal workstation to evaluate the basic algorithm and to explore several optimizations. We began by indexing the clauses using several keys to avoid searching all clauses during the subsumption check, the reduction step and the instantiation step.

To speed up the subsumption check, a *complete key* based on the predicate names and their arities is created. For example, the clause:

[3] By *length* we mean number of symbols, ignoring the number of characters in any given symbol.

```
p(X,Y) ← q(a,X,Y), r(f(X,Y)).
```

has the complete key **p-2-q-3-r-1**. To determine whether a clause is subsumed by any existing derived clauses, only the clauses in the clause database with the same complete key need to be considered.

For reduction, we maintain an index based on the *selected-literal key* which consists of the predicate name of the selected (i.e., first) literal and its arity. The selected-literal key for this clause is **q-3**. Each unit clause is used in an attempt to reduce all other clauses in the database. Since the unit clause must unify with the selected literal of other clauses, only those clauses with a selected-literal key matching the complete key of the unit clause need to be retrieved.

Finally for instantiation, an index based on the *program-rule-head key* is maintained. For every non-unit program clause, a program-rule-head key is computed from the predicate name of the head (positive) literal and its arity. For the above clause, it is **p-2**. Given a non-unit derived clause that we wish to use to instantiate program clauses, we compute a key based on its selected literal and arity. Then we need only consider those clauses with an identical program-rule-head index.

## Optimizations for DATALOG Programs

To increase the algorithm's performance further while restricting it to DATALOG programs, secondary indices based on *format vectors* are maintained in addition to the primary indices described above. The format vector for a clause is a string containing information about which argument positions are filled by constants and about variable usage in the clause. The format vector for the clause:

```
p(a,X,Y) ← q(Y,b), r(X).
```

is **#-1-2-2-#-1**. The predicate and arity information (which is contained in the primary keys) is not present in the format vector. The character "#" appears in the format vector in positions corresponding to constants and the numbers serve as normalized variable names.

Given the complete key and format vector for a clause, all that is needed to fully specify the clause (up to renaming of variables) are the values of the constants, which are represented simply as tuples. In a database with more than a couple of clauses that are equal up to constant values (and renaming of variables), this rather complex data representation saves space. Since many clauses with identical keys and format vectors are generated during a typical DATALOG execution, this representation pays off.

The main optimization for DATALOG, however, is *compiling* the reduction and instantiation steps. When a new clause is generated, it becomes necessary to compare it with all existing clauses to see what new clauses can be derived using the reduction or instantiation rules. Given such a *candidate clause*, we must look through the primary indices and, for each, we must look through all format vectors. Associated with each of these is a set of tuples, each one representing a clause. Since all these tuples (clauses) have the same key and same format vector, the

unification can be done for all the tuples at once by abstracting away from the actual values of the constants. The result of such a compiled unification is a sequence of equality checks, which can then be evaluated quickly for each of the tuples in the set.

We gloss over the details of the compilation step (see [Porter, 1985]) by giving an example compilation for a reduction step. Consider the candidate clause:

```
q(a, b, b, U, U, V, V).
```

This clause must be used to reduce all clauses with a seven-placed predicate named `q` as the selected literal. To find these clauses, we first use the selected-literal index to retrieve all those clauses with keys of the form *x*-*x*-**q-7**-*x*-*x*-**....** One such key is **p-3-q-7-r-3** and it will be used for this example.

Associated with this key are several format vectors. We will look at

**1-2-#-#-2-2-#-#-3-4-4-#-2**

For example, the clauses

```
p(W,X,a) ← q(b,X,X,c,d,Y,Z), r(Z,e,X).
p(U,V,a) ← q(a,V,V,c,c,Y,W), r(W,e,V).
p(W,U,a) ← q(b,U,U,d,d,Y,Z), r(Z,e,U).
```

would be represented by the tuples

```
a b c d e
a a c c e
a b d d e
```

We call these tuples (clauses) the *target* tuples (clauses).

The compilation phase may fail, in which case we know that none of the target tuples unify with the candidate tuple without ever looking at any of the target tuples. In this example however, the compilation succeeds producing the following "instruction" sequence:

$\#_2$ = `a`

$\#_3 = \#_4$

These instructions say that any tuple with the constant `a` in the second position and with the third and fourth positions equal unifies with the candidate clause.

For every such tuple we must construct a new derived tuple. By removing the selected literal from the target clauses, we get the key **p-3-r-3**. The compilation phase also produces a format vector describing the new clauses (**1-#-#-2-#-#**) along with the following information telling how to construct the new derived tuples from the target tuples:

$\#_1 \leftarrow$ b
$\#_2 \leftarrow \#_1$
$\#_3 \leftarrow \#_5$
$\#_4 \leftarrow$ b

After the compilation is complete, the equality comparisons are evaluated for each of the target tuples. Only the second tuple satisfies them. The following derived tuple can then be constructed using the tuple creation information:

```
b a e b
```

This tuple represents the desired clause:

```
p(X,b,a) ← r(Y,e,b).
```

These operations—comparing and manipulating tuple values—are familiar from relational algebra [Maier, 1983]. In fact, the process of executing the comparisons and creating new tuples representing reduced clauses for any tuples found to satisfy the comparisons can always be expressed using standard relational operators. If we label the positions of the target tuples with the attribute names $A_1$, $A_2$, ... $A_5$ and call the set of target tuples the relation r, the set of tuples representing the reduced clauses in this example can be represented (using the notation of [Maier, 1983]) as:

$$\delta_{A_2,A_3 \leftarrow A_1,A_5} (\pi_{\{A_1,A_5\}}(\sigma_{A_2=a} \ (r[A_3=A_4]r))) \bowtie <b{:}A_1\ b{:}A_4>$$

A very similar compilation-execution technique is used to speed up the instantiation step and the subsumption check.

Earley Deduction will terminate for DATALOG programs even if the subsumption check is relaxed to an equality check. In that case, a new tuple is not added to the derived set if it is already there. By using a hash table index for the individual tuples, this check can be done in essentially constant time. We will save time only if the time saved by using the equality check outweighs the additional time associated with processing extra clauses that would have been deleted by the full subsumption check.

Another optimization we implemented involves batching up the subsumption checking. In the course of a reduction (or instantiation) step, a number of clauses with identical keys and format vectors will be created. The subsumption check must be performed on each of these before it can be added to the derived set. By delaying the subsumption checking until one of these tuples is referenced, a number of very similar compilations can be replaced with a single compilation. Then the subsumption check for all of the clauses is performed at one time by repeatedly executing the compiled “instructions”.

## Conclusions

All of the implementation optimizations described above were implemented and a number of programs were executed to determine whether and how much they speeded up the Earley Deduction algorithm. For general logic programs, Earley Deduction is not nearly fast enough to compete with typical Prolog interpreters. Its usefulness arises when you want to run logic programs without concern for the order of the program clauses (or literals within the clauses) and/or you want all solutions in the face of possible non-termination. We are interested in using it to execute large Natural Language rule-based parsers. We want to be able to express the grammar as clearly as possible, without letting implementation details like clause/literal order get in the way. As in many logic programs, the rules in logic grammars express general knowledge about language and it is often difficult to foresee how they will be used. Another area where the generality of Earley Deduction is desirable is for systems in which the clauses are generated automatically by a program that would be unnecessarily complicated by concern about execution order.

Representing the clauses as tuples and compiling the reduction step, the instantiation step and the subsumption check for DATALOG programs resulted in a significant speed-up over the general algorithm—over 10 times for some of the programs tried. Replacing the subsumption check with the simpler equality check speeded the algorithm up a little more (8.8% for the programs we tried) and replacing the equality check by the batched subsumption check improved performance even more (20.3%). Furthermore, the improvement realized from the DATALOG optimizations increases as the length of the deduction grows since compiling has a greater benefit the more times the compiled instructions are executed.

Our experiments were performed entirely within the Smalltalk environment. Using the clause representation described above, it would be fairly straightforward to store the tuples in a traditional relational database, keeping the compilation and overall system organization in Smalltalk. All access to the tuples can be done using standard relational operators. In this way, the system could be enhanced to handle large DATALOG programs. An empirical evaluation of such a system (vis-à-vis a Prolog interpreter) is one direction for further research.

## Related Work

The most closely related work has been done by S. W. Dietrich and D. S. Warren. They describe a method for executing logic programs using *extension tables* which is also complete for DATALOG programs [Dietrich & Warren, 1986]. Their approach is to apply dynamic programming to the execution of logic programs. In the basic algorithm, a table of computed tuples is associated with each predicate. Each time a rule is used to compute a new tuple for a predicate, it is added to the corresponding table—thus the name *extension tables*. Certain modifications are necessary to avoid infinite looping in the presence of recursive rules. The use of extension tables is particularly exciting because it can be applied to nasty predicates without invoking the overhead of large extension tables for trivial predicates which are easier to recompute when needed.

D.E. Smith, M.R. Genesereth and M.L. Ginsberg define a recursive inference as an infinite sequence of goals containing repeated similar subgoals. By examining goal sequences, certain portions of the search space can be identified as unable to produce any new answers. They present a number of theoretical results about where the search for a proof may safely be pruned [Smith, Genesereth & Ginsburg, 1986]. In addition, they also present a number of provocative examples while discussing infinite inference chains and techniques for dealing with them.

Another system concerned with the execution of functor-free programs is Jeffrey Ullman's NAIL! system [Ullman, 1984, Ullman, 1985]. Summarizing, the system compiles queries by building a *rule/goal graph* describing the clauses and their interconnectivity. The system then analyses this graph trying to find a relational expression that effectively computes the answer relation. The desired relational expression can be produced if nodes in the tree can be *captured* and a number of *capture rules* are described. L. J. Henschen and S. A. Naqvi also describe an algorithm for compiling queries in the functor-free subset of logic [Henschen & Naqvi, 1984].

## Summary

In summary, we have described Earley Deduction, an algorithm for executing logic programs which appears useful in specialized applications, particularly where non-termination is a problem. We then looked at the relational subset of logic and discussed a number of implementation techniques that improved performance considerably in our experiments. It appears that, for sufficiently large programs in the relational subset, Earley Deduction can be made faster and more practical than the traditional Prolog interpreter.

## Acknowledgements

I am gratefully indebted to the following people, with whom stimulating conversations have led to the ideas presented here: David S. Warren, Fernando Pereira, Susan W. Dietrich, Mark Grossman, Mark Ballard, Cliff Walinsky and, particularly, David Maier.

## REFERENCES

[Dietrich & Warren, 1986]
Dietrich, S.W. and Warren, D.S., Extension Tables: Memo Relations in Logic Programming, Technical Report 86/18, CS Dept., SUNY, Stony Brook, New York, 1986.

[Earley, 1970]
Earley, Jay, An efficient context-free parsing algorithm, *Comm. ACM* 6(8):451-455 (1970).

[Henschen & Naqvi, 1984]
Henschen, L.J. and Naqvi S.A., On Compiling Queries in Recursive First-Order Databases, *J. ACM* 31(1):47-85 (1984).

[Maier, 1983]
Maier, David, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[Nilsson, 1971]
Nilsson, Nils J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

[Pereira & Warren, 1983]
Pereira F.C.N. and Warren D.H.D., Parsing as deduction, *ACL Conference Proceedings*, 1983.

[Porter, 1985]
Porter, Harry H. III, Optimizations to Earley Deduction for DATALOG Programs, Unpublished memorandum, 16 October 1985.

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

[Smith, Genesereth & Ginsburg, 1986]
Smith, D.E., Genesereth, M.R. and Ginsberg, M.L., Controlling Recursive Inference, *Artificial Intelligence*, 30(3):343-389 (1986).

[Ullman, 1984]
Ullman, J.D., Testing Applicability of Top-Down Capture Rules, Unpublished memorandum, Dept. of CS, Stanford University, 1984.

[Ullman, 1985]
Ullman, J.D., Implementation of Logical Query Languages for Databases, *ACM Trans. Database Systems*, 10(4), (1985).