Earley Deduction

Harry H. Porter, III

Technical Report No. CS/E 86-002

March 10, 1986

Earley Deduction

Harry H. Porter, III

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 86-002
March 10, 1986

Earley Deduction

Harry H. Porter, III
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006-1999

Technical Report CS/E-86-002
March 10, 1986

Abstract

This paper first reviews Earley deduction, a proof method appropriate for PROLOG execution, having advantages over the execution rule used by the standard PROLOG interpreter. One advantage is that correctness and, unlike the standard interpreter, completeness are guaranteed. Informal proofs of this are included. Another advantage is that, for a restricted class of programs known as DATALOG programs (i.e. PROLOG without functors), it is guaranteed to terminate (unlike the PROLOG interpreter). An informal proof of this is also given. As initially described, the algorithm does not appear to be fast enough to be useful. We conclude by describing implementation techniques that make Earley Deduction practical for DATALOG programs.

Earley Deduction

There are two general schemes for parsing context-free grammars: top-down and bottom-up. Top-down is goal driven. The parser begins by asking what non-terminal is sought and works by then asking what possible non-terminals can be found to satisfy the higher-level goal. Bottom-up is data driven. Here the parser looks to see what terminals are present and asks what non-terminals can be composed out of them. Both methods have their advantages and disadvantages. The top-down parser may fail to find a valid parse by failing to terminate. The bottom-up method may do more work than necessary.

Jay Earley devised a method, call *Earley Parsing* or *chart parsing*, that is a combination of top-down and bottom-up [Earley 1970]. It is goal-driven since the method never looks for any non-terminal unless it is needed and it is bottom-up in the sense that it saves partial results so that they may be re-used instead of being recomputed.

The traditional PROLOG interpreter is top-down. It begins with a goal and subsequently refines it. Like a top-down parser, it is depth-first, trying first one rule and if that fails, backing up to try another. As PROLOG programmers well know, if the program contains left-recursive rules, the interpreter may go into an infinite loop, failing to find a proof even though one exists. Bottom-up strategies have also been investigated. With DATALOG programs (i.e. those PROLOG programs which contain no functor symbols) the bottom-up procedures will terminate but, just as for context-free parsing, they may do a lot more work than necessary.

F.C.N. Pereira and D.H.D. Warren have extended Earley Parsing to the execution of PROLOG programs and call the method *Earley Deduction* [Pereira and Warren 1983]. We will explain the method by tracing its execution on an example PROLOG program taken from their paper. Later, we will show that, for DATALOG programs, this algorithm always terminates.

The clauses that make up the program are:

$$p(X,Z) \leftarrow p(X,Y),\ p(Y,Z). \qquad (1)$$

p(a,b). (2)
p(b,c). (3)

The first clause is a rule with the positive literal on the left-hand side and the negative literals on the right-hand side. The second and third clauses are called unit clauses since they have no clauses on the right-hand side. The method uses a dummy predicate *ans* for program goal clause. The program goal 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*. Let's look at a reduction first. Consider clause (4) and unit clause (2). Clause (4) says that we want to show p(a,Z). A one-step solution for this program can clearly be made using clause (2) giving an answer with Z=b. We can thus add

---

ans(b). (5)

---

to the set of derived clauses.

Each step (both instantiation and reduction) works by selecting a derived clause and combining it with another clause (either program or derived). The former will be called the *selected clause*. Within the selected clause, there will be a *selected literal*. It can be chosen when the

clause is first created and added to the derived clause set or at the time the clause is first chosen as the selected clause. We will always choose the first literal as the selected literal.

To be more specific, the reduction step works as follows. First, unify the selected literal of the selected clause with a unit clause. The selected clause must always be a derived clause but the second clause – the unit clause – can be either a program or a derived clause. Let σ be the most general unifier. In the example, p(a,Z) is the selected literal of the selected clause since it is the left-most literal on the right-hand side. The unifier is $\sigma = \{ Z \leftarrow b \}$.

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

The second kind of inference rule is called *instantiation*. To illustrate this rule, we will use clause (4) as the selected clause to instantiate clause (1). Clause (4) says that we can show ans(Z) if we can show p(a,Z). Clause (1) is a rule that can be used to show p(X,Z) and can be used. The clause that we add to the derived set is

---

p(a,Z) ← p(a,Y), p(Y,Z). (6)

The instantiation step says to take the selected literal of the selected clause (a derived clause) and unify it with the positive (left-hand side) literal of a non-unit program clause, giving a most general unifier σ. The unification of the selected literal p(a,Z) with the left-hand side literal of clause (1) gives $\sigma = \{ X \leftarrow a \}$. Then apply σ to the program rule (clause (1)) and add the result as a new derived clause.

Clause (6) now has two negative (right-hand side) literals and the first, p(a,Y), is the selected literal. Since it unifies with the unit-clause (2), we can perform a reduction step and add:

---

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 ground 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.

There are several more instantiations and reductions we can perform before we reach a point where no new clauses can be derived. Below, we complete this example, leaving the reader to hand-trace each step if desired. We have included comments and, for convenience, the previously listed clauses are repeated.

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)

Discussion

We can informally understand how the procedure works by recognizing that the derived clauses are true statements vis-a-vis the program. Derived unit-clauses, such as clause (9), represent true (but not necessarily ground) facts. Derived rules, such as (10), represent true rules. Clause (10) was produced from a previous rule (clause (8), inductively assumed to be true) by using clause (3), also assumed to be true. Thus we must conclude that clause (10) is true. In this way, the algorithm exhibits a bottom-up character.

Furthermore, the right-hand sides of derived non-unit clauses (including those clauses with ans(...) as their heads) represent goals we have discovered that need solving in order to produce an answer. We start with one (clause (4)) that contains the original goal on its right-hand side. The right-hand side of clause (7) indicates that we need to solve p(b,Z) in order to produce the answer p(a,Z). Each new derived non-unit clause comes from an existing subgoal. Either we reduce the subgoal by removing its selected literal in a reduction step (using a unit-clause telling us the selected literal is true) or we use a program non-unit clause to generate a new subgoal. In this way, the algorithm works top-down.

Correctness of Earley Deduction

We next give a very informal argument that this proof procedure is *correct* in the sense that any answer obtained implies the query is a logical consequence of the program. New clauses are added to the bottom of the derived set, implying they rest on previous clauses. We assume the reader is familiar with refutation proofs [Robinson 1965] and show that all clauses are true inductively on the sequential numbering of the clauses.

Recall that a PROLOG 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 rewritten 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 ans(...) denotes the empty clause and also carries information about the binding that was used in deriving it.

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 a program rule. Thus, if the unit clause ans(...) is derived, the empty clause has been produced and thus the query is proved true. 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.

Completeness of Earley Deduction

We will call trees like the one in Figure 1 *Earley Deduction Trees*. The tree will always have the empty clause ans(...) at the root and every node will either (1) have 2 children or (2) be a program clause, goal clause or derived unit clause. If the node has 2 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 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 defined below) exists [although the PROLOG interpreter won't 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 rule 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 substitution happened to eliminated all variables but that won't always happen.

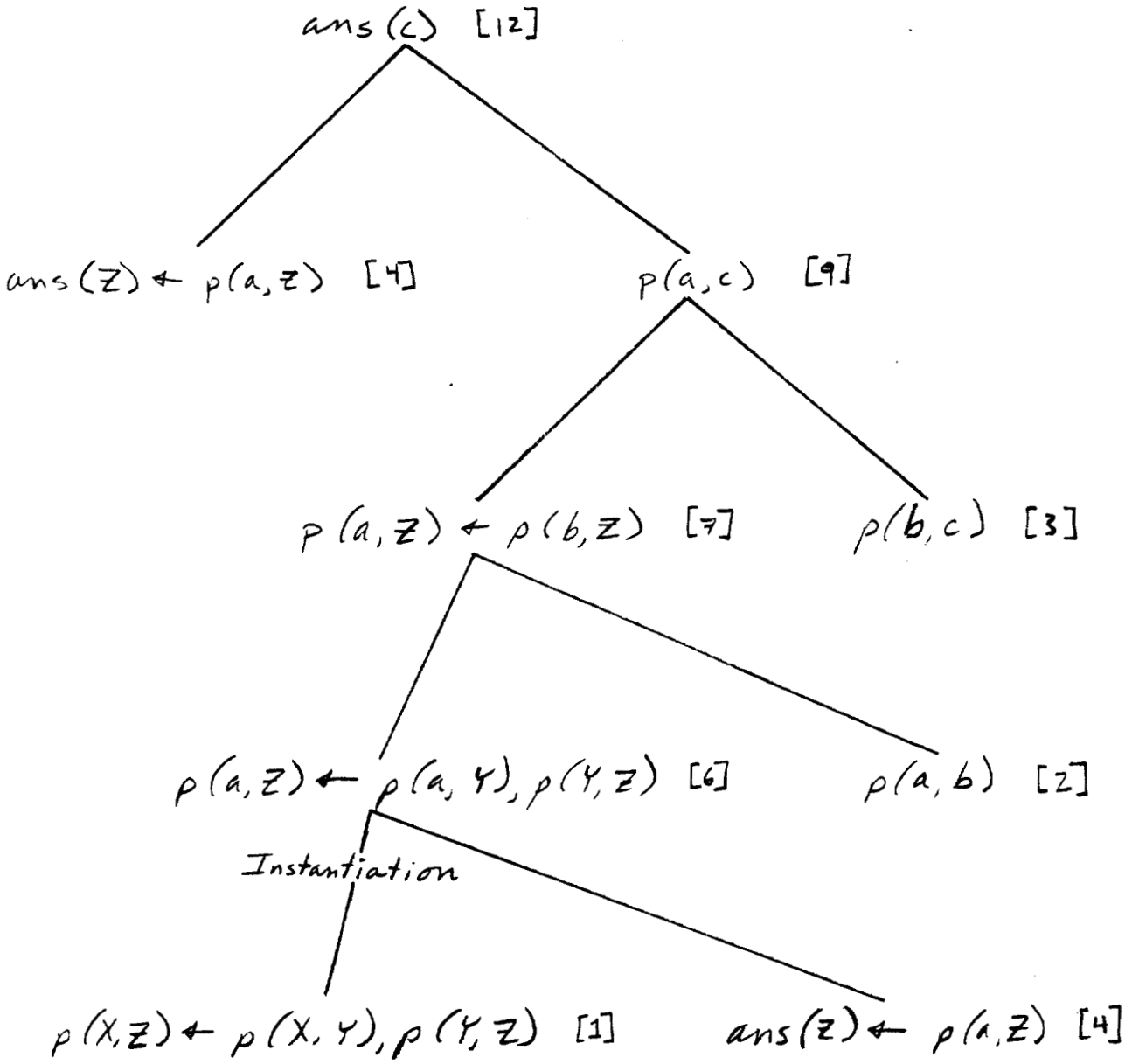

Figure 1

Also note that the head of a child clause exactly matches a negative literal in its parent's right-hand side. If the query is provable, then such a tree exists.

Next, we show how to construct an Earley Deduction Tree from a PROLOG proof tree. Call the result the *constructed tree*. With every node in the PROLOG tree associate an Earley

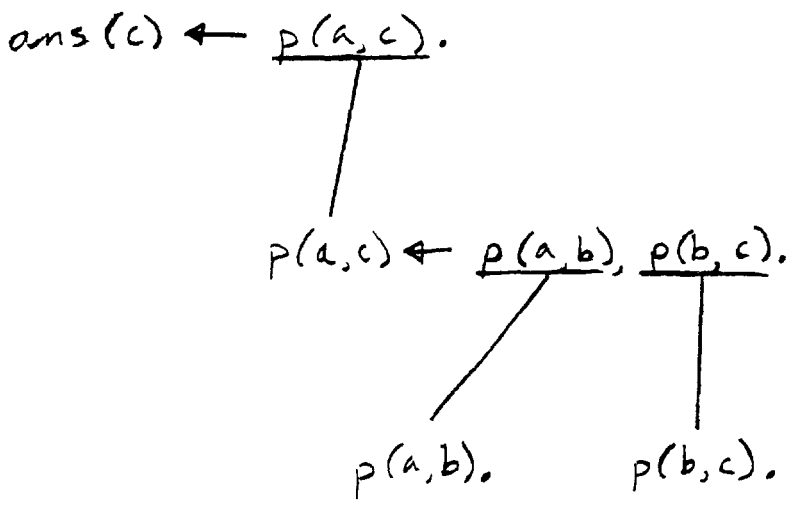

Figure 2

tree. The construction is defined recursively, starting at the leaves of the tree and working up to 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. Translating interior nodes of the PROLOG trees is a little trickier and is shown diagrammatically in Figure 3. Figure 4 shows the constructed tree we get from the PROLOG proof tree shown in Figure 2.

Before taking the last step, we need to specify how the Earley algorithm selects pairs of clauses for combination, which we have not yet done. Not every selection strategy will find answers even when they exist, as the following example demonstrates:

Program Clauses:		
p(X) ← p(f(X)).		(14)
p(a).		(15)
Derived Clauses:		
ans ← p(a).	Goal	(16)
p(a) ← p(f(a)).	15 instantiates 13	(17)
p(f(a)) ← p(f(f(a))).	16 instantiates 13	(18)
p(f(f(a))) ← p(f(f(f(a)))).	17 instantiates 13	(19)
. . .		

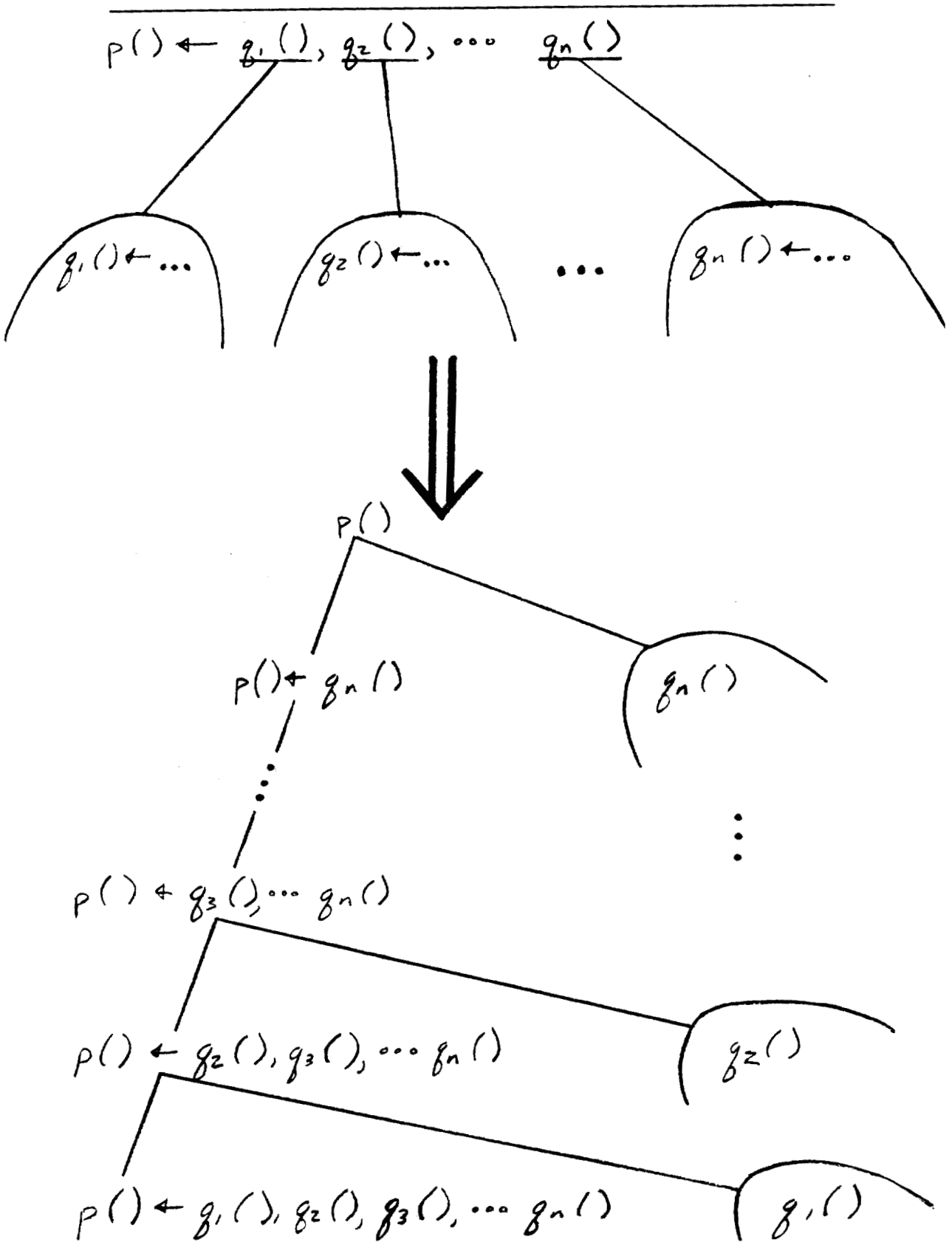

Figure 3

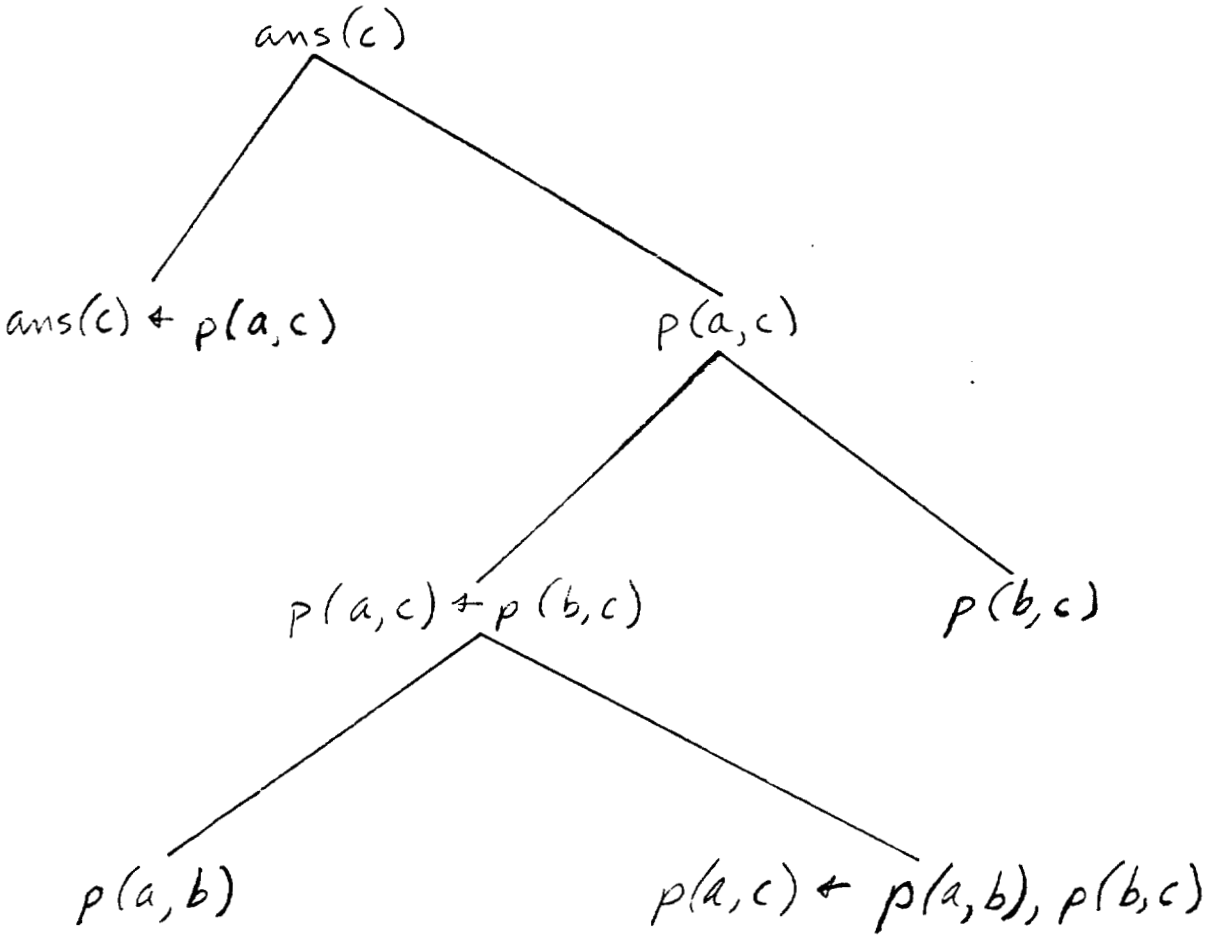

Figure 4

At some step in the algorithm let clauses **C1** and **C2** be two clauses that can be combined (either using instantiation or reduction) to produce a new clause **C3**. We will make the assumption that the selection strategy will eventually get around to combining **C1** and **C2** and considering **C3**. Eventually, a clause at least as general as **C3** will be added to the derived set (since **C3** itself will be added unless it is subsumed by some previously derived clause). Call such a selection strategy *fair*. [One strategy meeting the fairness assumption is the rule that says combine the oldest clauses first.]

Next, consider an interior node of the constructed Earley tree built from the PROLOG proof tree. It is labelled with clause **C3** and has two children whose roots are labelled with clauses **C1** and **C2**. Assume that the Earley deduction algorithm has already produced two clauses that are at least as general as **C1** and **C2**. Then, using the assumption about the combination strategy, they will ultimately be combined to produce a clause at least as general as **C3**.

Finally, we note that the Earley algorithm begins with a collection of program and goal clauses containing clauses at least as general as the clauses labelling 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 labelling the root of the PROLOG proof tree. Thus, assuming a fair selection strategy, Earley Deduction is complete and will eventually find all existing solutions.

Termination for DATALOG

A DATALOG program is just a PROLOG program that contains no functors. Our example was such a program. Does Earley Deduction always terminate for DATALOG programs? Yes. A rough justification follows.

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. 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 bigger clauses. Instantiation takes a program clause and instantiates it and so it can't be used to make a bigger clause either.

Assuming 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 (which is implicit), there are only a finite number of clauses with k or fewer literals using only the predicates and constants appearing in the program. (Two clauses differing only in the names of variables are

considered equal.)

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 the subsumption check insures that we never add a clause to the derived set if it is already there (since every clause subsumes itself), and since we can find new derived clauses (if they exist) in finite time, the procedure must eventually derive all derivable clauses. Then, after examining all pairs of clauses looking for a new derived clause, the procedure will find none and terminate. Of course, when we allow functor symbols, the derived clauses may be bigger than either of the two existing clauses and so the procedure is not guaranteed to terminate.

Implementation

We have implemented Earley Deduction using Smalltalk on the Tektronix 4404 personal workstation to evaluate the speed of the basic algorithm and to explore several optimizations. We begin 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:

---

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, we only consider clauses in the clause database with the same complete key.

For reduction, we maintain an index based on the *selected-literal key* which consists of the predicate name of the first literal and its arity. The selected-literal key for this clause is **q-3**. Given a unit clause, we will use it to reduce all clauses in the database. Since the unit clause must unify with the selected literal of other clauses, we need only retrieve those clauses with a selected-literal key matching the complete key of the unit clause.

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. These are represented simply as tuples of constant values. In a database with more than a couple of clauses that are

equal up 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 this 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. 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 $A_1, A_2, \cdots A_5$ and call the set of target tuples the relation r, the set of tuples representing the reduced clauses in this example can be represented in the notation of [Maier 1983] as:

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

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

Earley Deduction is guaranteed to 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. This 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. It usefulness lies where one desires to run logic programs without concern for the order of the program clauses or where one wants all solutions for programs that do not terminate. 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 order get in the way. The grammar rules 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 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 using Smalltalk which placed limits on the size of programs our system can handle. The clause representation we use would make it fairly straightforward to store the tuples in a traditional relational database, keeping the compilation and overall system organization in Smalltalk. The accesses made to the tuples can all be done using standard relational operators. In this way, our system could be enhanced to handle large DATALOG programs and we believe a comparison between such a system and PROLOG interpreters is the logical next step.

In summary, Earley Deduction in general appears too slow for the execution of logic programs in all but very specialized applications. However, when the logic program does not contain any functors, enough improvements can be made in its performance to make it feasible. Furthermore, it appears that for sufficiently large DATALOG programs, it can be made faster than the traditional PROLOG interpreter.

References

Earley 1970
Earley, Jay, An efficient context-free parsing algorithm, CACM 6:8, 1970, p.451-455

Maier 1983
Maier, David, *The Theory of Relational Databases*, Computer Science Press, 1983

Pereira and Warren 1983
Pereira F.C.N., 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 draft, 1985

Robinson 1965
Robinson, J.A., A Machine-Oriented Logic Based on the Resolution Principle, JACM 12:1, January 1965, p.23-41