Complexity of Pure Prolog Programs

Erik Aarts[1]

Research Institute for Language and Speech
Trans 10
3512 JK Utrecht
The Netherlands

Dept. of Mathematics and Computer Science
Plantage Muidergracht 24
1018 TV Amsterdam
The Netherlands

Abstract

This paper gives a method to estimate the space and time complexity of algorithms implemented in pure Prolog and executed under the OLDT search strategy (also called Earley Deduction).

1 Introduction

This article is about the complexity of Prolog programs. The worst case time complexity of programs written in an imperative language (like Pascal or C) can be estimated by straightforward means. These programs are deterministic so we can follow the execution step by step. The number of steps is counted by estimating the cost of smaller procedures, e.g. multiplying the number of times that a "while" loop is executed with the number of steps needed in every loop etcetera. A disadvantage is that the code of larger programs gets incomprehensible very soon. This is solved by presenting pseudo-code. In pseudo-code, however, the reader has to guess the details. In fields like computational linguistics and artificial intelligence we often see algorithms explained with "real" Prolog code. This can be done because real Prolog code is easier to read than e.g. C or Pascal code. An algorithm presented this way, has no open-ended details. For Prolog programs, however, the complexity analysis is not so easy. The main problem is that Prolog programs are intended to be non-deterministic. Computers are deterministic machines, however. Therefore any Prolog interpreter has to deal with the non-determinism in some deterministic way. Standard interpreters perform a depth-first search through the search space. In case of a choice point the interpreter takes a decision. If that decision appears to be wrong later, the interpreter reverses the decision and tries another possibility. This mechanism is called backtracking. Backtracking

[1]The author was sponsored by project NF 102/62-356 ('Structural and Semantic Parallels in Natural Languages and Programming Languages'), funded by the Netherlands Organization for the Advancement of Research (NWO).

is what makes the analysis of Prolog programs so hard. The only attempt to estimate the runtime of Prolog programs in the context of standard interpreters is from Lin (1993). This is a fairly complex method however.

This article does not solve the problem of estimating the runtime for standard interpreters. However, we can give runtime estimates easier than in (Lin 1993) if we use an interpreter called the *Earley interpreter*. The Earley interpreter does not backtrack, but it keeps an administration of what things have been tried and what the result was. It differs in two ways from the standard interpreter:

- improved proof search. Prolog programs have a very clear meaning from a logical point of view. Standard interpreters do not behave properly however. They sometimes don't find a proof although there exists one, because they get stuck in an infinite loop. One can "program around" this but then we leave the path of declarative programming. The Earley interpreter does what it should do. It can only get in a loop if terms can grow arbitrarily long.

- longer runtime. Because the interpreter has to do a lot of bookkeeping the runtime will be longer in general. This is the major disadvantage of the method presented here: in order to estimate the runtime we use an interpreter that increases the runtime. Lin (1993) does not have this disadvantage. There are two arguments in favor of this method. First the bookkeeping can speed up algorithms too. It can even speed up an exponential time algorithm to a polynomial time algorithm. Second, the overhead is small, usually linear in the size of the input. This means that we stay in the same complexity class in most cases.

The main reason to switch from the standard interpreter to the Earley interpreter is the possibility to prove runtime bounds for Earley interpreters in a pretty straightforward way. We will describe a simple method to deduce the runtime of an algorithm from two sources: the length of the instantiations of the variables in the program and the number of possible instantiations. If a Prolog programmer knows how the variables in his program behave, he can deduce the runtime in a simple manner.

The main idea behind our approach is the following. The Earley interpreter stores all attempts to prove something (i.e. it stores all "procedure" calls). Furthermore it stores all solutions (and all partial solutions). Because of this we are sure that every procedure is executed only once for every assignment to the variables. When the procedure is called a second time the answer can be looked up and this costs only very little time. This is called *memoization* or *tabulation* or *tabling*. The search strategy is called Earley Deduction. The Earley Deduction proof procedure is due to Warren (1975). It was first published in (Pereira and Warren 1983). A good introduction is (Pereira and Shieber 1987, pp. 196-210). Similar ideas can be found in (Warren 1992), (Tamaki and Sato 1986) (OLDT resolution) and (Vieille 1989) (SLD-AL resolution).

The fact that all problems are solved once makes it much easier to estimate the time complexity: we only have to count the number of procedure calls multiplied with the amount of time spent in the procedure for each call.

The structure of this article will be as follows. First we give a short introduction to Prolog. We describe the language and show what the logical (or declarative) meaning is. Then we describe a non-deterministic interpreter that does exactly what should be done according to the declarative meaning. Then we show two methods to make the interpreters deterministic. The first one leads to the standard interpreter. The second method to get a deterministic interpreter leads to the Earley interpreter.

When it is clear how the Earley interpreter works we start our complexity analysis. The result of the counting will be a complexity formula in which one has to fill in the length and the number of all possible instantiations for the variables. After this we describe shortly a speed up of the interpreter. We will sketch some ideas about further research. Finally we will say something about existing implementations of Prolog interpreters which are able to follow the search strategy we describe here.

2 Pure Prolog

2.1 Introduction

Prolog stands for programming in logic. Although it has its roots in mathematical logic we will introduce it here as a programming language and neglect the mathematical foundations. The introduction given here is based on (Bratko 1990). Conventional programming languages are procedurally oriented. Prolog introduces a declarative view on programming. One has to give a description of the problem one wants to solve. Once this description has been given an interpreter will solve the problem. Ideally, the programmer should not bother about the way the interpreter solves the problem. This is the procedural part of the problem and it should be "hidden". However, in real life it is not always possible to separate the description of the problem and the search for solutions.

The reason for this is that that the declarative meaning and the procedural meaning of Prolog programs often differ. A Prolog program consists of a set of axioms (or unit clauses) and a set of rules. The declarative meaning is defined as follows. Some expression is true if it can be derived from the axioms via the rules. When the expression is not derivable it is false. The declarative meaning is what the interpreter *should* do. The procedural meaning is what the interpreter in fact does. I.e., something is true if the interpreter *can find* a derivation, and it is false if it *can not find* a derivation (and terminates).

It should be clear that the procedural and declarative meaning of Prolog programs coincide. If we want to implement an Earley interpreter we face two choices. We have to decide whether we want to follow a breadth-first strategy or a depth-first strategy. Furthermore, we can choose between an exhaustive search, that generates all solutions, or a non-exhaustive search, which generates only one solution. The interpreter that performs a breadth-first non-exhaustive search is *complete*, i.e., it will find a proof if there is one.

The Earley interpreter has more advantages than the improved proof search. It is much easier to give estimates for the complexity of problems. The solution of a problem is given by a declarative description plus some interpreter that finds proofs. The complexity of the problem is the number of steps that the interpreter takes in finding proofs (or in finding out that there are no proofs).

2.2 Definitions

In this section we will give definitions of the syntax of programs and of the declarative (non-procedural) meaning. In the next section we will discuss the procedural meaning.

The language has a countable infinite set of *variables*, and countable sets of *function* and *predicate symbols*. Each function symbol f and each predicate symbol p is associated with a natural number n, the *arity* of the symbol. A function symbol with arity 0 is referred to as a constant. A *term* in the language is a variable, a constant or a *compound term* $f(t_1, \ldots, t_n)$, where f is a function symbol with arity n, and t_1 through t_n are terms. A term is *ground* if it contains no variable.

We use the convention that variables are written with an uppercase-letter. Constants, predicate and function symbols start with a lower-case letter. Terms represent complex data structures. E.g. the date "May 1 1993" can be represented as `date(1,may,1993)`. Any date in that same month can be represented as `date(Day,may,1993)`.

An *atom* $p(t_1, \ldots, t_n)$ consists of an n-ary predicate symbol p and n terms t_i as the arguments of p. An atom is *ground* if all its arguments are ground terms. There are three types of *Horn clauses*: *facts, rules and queries.*

- Rules declare things that are true depending on some conditions. They are written as p `:-` $q_1, \ldots, q_n$, where p, q_i are atoms. Read this as: p if q_1 and ... and q_n. The atom p is called the *head*, and $q_1, \ldots, q_n$ is the body.

- Facts declare things that are always, unconditionally, true, i.e. they are rules with an empty body. They are written as p., where p is an atom.

- By means of queries the program can be asked what things are true. A query is of the form `?-` $q_1, \ldots, q_n$, where all q_i are atoms.

A *definite clause* is a rule or a fact. A *predicate definition* consists of a finite number of definite clauses, all with the same predicate symbol in the head. A *logic program* consists of a finite number of predicate definitions.

Compound terms and atoms are sometimes written in the infix notation instead of the prefix notation. A function symbol often used that is not prefix is the list constructor `[H|T]`. When we try to prove an atom, we sometimes say that we try to prove a *goal*. A function symbol is also called a *functor*.

With a definition of facts and rules we can define truth values for queries. This is called the declarative meaning of Prolog programs.

A *substitution* is a mapping from variables to terms that is the identity mapping at all but finitely many points.

Definition 2.1 *A query* `?-` $Q_1, Q_2, \ldots, Q_n$ *is true iff there is a substitution* σ_1 *such that the atoms* $\sigma_1(Q_1)$ *and ... and* $\sigma_1(Q_n)$ *are true.*

An atom G *is true if and only if there is a clause* C `:-` $B_1, B_2, \ldots, B_n$ *in the program* ($B_1, \ldots, B_n$ *can be empty, then* C *is a fact) and a substitution* σ_2 *such that* $\sigma_2(C)$ *is identical to* G *and the atoms* $\sigma_2(B_1)$ *and ... and* $\sigma_2(B_n)$ *are true.*

Often there are infinitely many substitutions that make a query true. Then we are interested in the most general substitutions. A substitution σ_1 is more general than a substitution σ_2 if there is a substitution σ_3, not the identity, such that $\sigma_2 = \sigma_3 \circ \sigma_1$, where $\circ$ is the function composition operator. A substitution σ is a solution for a query if the substitution makes the query true and there is no more general substitution that makes the query true. But even then a query can have infinitely many solutions. If there is no solution the query is false.

3 Earley Provers

When we have the declarative meaning as defined in Definition 2.1 in mind, a non-deterministic prover can be given that follows the declarative meaning precisely. The prover answers yes if there is a proof (but not vice versa). Before the introduction of the prover we first define *unification*.

We say that a term Z is a *unifier* for the terms X and Y if there is a substitution σ such that $Z = \sigma(X)$ and $Z = \sigma(Y)$. We say that term X is *more general* than term Y if there is a substitution σ' such that $Y = \sigma'(X)$ and $Y \neq X$. A term Z is the *most general unifier (mgu)* of X and Y if it is a unifier of X and Y and there is no unifier that is more general. The most general unifier is unique modulo variable renaming. Two terms T_1 and T_2 are *alphabetic variants*, if there are substitutions σ_1 and σ_2 such that $T_1 = \sigma_1(T_2)$ and $T_2 = \sigma_2(T_1)$. All these notions can be defined on atoms instead of terms too. There exist efficient unification algorithms, that are linear in the size of the terms (Paterson and Wegman 1978).

The non-deterministic prover is in Figure 1.

The only non-determinism in this prover is the guessing of a clause. In standard provers this non-determinism is eliminated by performing a depth-first search. Every time the prover has to guess a clause it takes the first one available. If it turns out later that this choice was wrong, i.e. that no proof can be found, we try the second possibility and so on.

Prove list of goals:

Given a list of goals $Q_1, \ldots, Q_n (n > 1)$, prove Q_1. The result will be a substitution σ_1 such that $\sigma_1(Q_1)$ is derivable. Then prove the list of goals $\sigma_1(Q_2), \ldots, \sigma_1(Q_n)$. The result of proving $\sigma_1(Q_2), \ldots, \sigma_1(Q_n)$ is a substitution $\sigma_n \circ \ldots \circ \sigma_2$. The result of proving $Q_1, \ldots, Q_n$ is $\sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$.

Prove goal:

Given a goal G, guess a clause C :- $B_1, B_2, \ldots, B_n$ in the program and compute $mgu(G, C)$. Suppose $mgu(G, C) = f(C) = g(G)$. Prove the list of goals $f(B_1), f(B_2), \ldots f(B_n)$. The result is the substitution h. The result of proving goal G is the substitution $h \circ g$.

Figure 1: Non-deterministic prover

This strategy often leads to problems. Consider the following program. It computes the reflexive transitive closure of a graph.

```
% Sample program PATH

path(X,Z) :-
        path(X,Y),
        edge(Y,Z).
path(X,X).

edge(a,b).
edge(b,c).
edge(c,a).
edge(c,d).
```

Suppose we have a query `?- path(a,d).` A depth-first searching Prolog prover will try to prove the following goals: `path(a,d)`, `path(a,Var1)`, `path(a,Var2)`, `path(a,Var3)`, `path(a,Var4)` etc. The prover will never get out of this loop.

Earley interpreters are defined as follows. The basic data structure we use is called the *item*. Items have the same form as atoms: they are pairs of heads and bodies. The head of the item is the head of some clause after some substitution. The body of the item is a (possibly empty) remainder of the same clause after the same substitution. Items are used to store partial results when proving a query. This is done as follows. Consider the prover in Figure 1. We have to prove some goal, and therefore we take an arbitrary clause from the program. After computing the *mgu* of the goal and the head of the clause, we obtain the item $\langle f(C), [f(B_1), f(B_2), \ldots, f(B_n)]\rangle$. Now we try to prove $f(B_1)$. This gives us the substitution σ_1 and the new item $\langle \sigma_1(f(C)), [\sigma_1(f(B_2)), \ldots, \sigma_1(f(B_n))]\rangle$. We prove $\sigma_1(f(B_2))$, find σ_2, and obtain $\langle \sigma_2(\sigma_1(f(C))), [\sigma_2(\sigma_1(f(B_3))), \ldots, \sigma_2(\sigma_1(f(B_n)))]\rangle$. In every step the body becomes shorter. Finally, the body is empty. The final item $\langle \sigma_n(\ldots(\sigma_1(f(C))), [\,]\rangle$ is a solution for the goal we tried to prove.

The data structure the Earley interpreter uses has been described now. The control structure is as follows. We keep an *agenda* of items that wait to be processed and a *table* of items that have been processed. When we process an item from the agenda we first look whether it occurs in the table. If it does, we can simply discard it. If it does not occur in the table there are two possibilities:

- the body is empty. That means that the item is a solution. We combine the solution with the items in the table that are waiting for that solution. This gives us new items which are placed on the agenda again. This operation is called completion.
- The body is not empty. Two operations are executed:
  - prediction. The first element of the body is unified with the head of clauses in the program. New items are put on the agenda again.
  - completion. The first element of the body is combined with solutions in the table.

We will implement two provers: one that generates all solutions and one that stops after it has found the first solution. If we want to generate all solutions, then we stop when the agenda is empty. When we want only one solution, we can stop when the first solution appears in the table.

The algorithms sketched above can be implemented in Prolog as follows. The main predicate for the one-solution prover is `prove_one`. It is called with the goal we want to prove as argument. For the all-solutions prover the main predicate is `prove_all`. This predicate is called with the goal and with a variable that will be instantiated to the list of all solutions. We can change between depth-first and breadth-first behaviour simply by swapping two arguments in some `append` predicate.

The program clauses are of the form `Goal ::- Body` in order to separate them from the clauses of the prover. The program is given as a whole first and in little parts with comment later.

```
% Earley prover.
% needs: findall, member, append, numbervars.

?- op(1150,xfx,::-).

prove_all(Goal,Solutions) :-
        findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
        extend_items_all(Agenda,[],Table),
        findall(Goal,member(item(Goal,[]),Table),Solutions).

extend_items_all([],Table,Table).
extend_items_all([Item|Agenda1],Table1,Table2) :-
        memberv(Item,Table1),
        extend_items_all(Agenda1,Table1,Table2).
extend_items_all([Item|Agenda1],Table1,Table3) :-
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        append(Items,Agenda1,Agenda2),  % depth-first search
%       append(Agenda1,Items,Agenda2),  % breadth-first search
        extend_items_all(Agenda2,Table2,Table3).

prove_one(Goal,YN) :-
        findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
        extend_items_one(Agenda,[],Goal,YN).

extend_items_one(_,Table,Goal,yes) :-
        member(item(Goal,[]),Table).
extend_items_one([],Table,Goal,no) :-
        \+ member(item(Goal,[]),Table).
extend_items_one([Item|Agenda1],Table,Goal,YN) :-
        \+ member(item(Goal,[]),Table),
        memberv(Item,Table),
        extend_items_one(Agenda1,Table,Goal,YN).
extend_items_one([Item|Agenda1],Table1,Goal,YN) :-
        \+ member(item(Goal,[]),Table1),
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        append(Agenda1,Items,Agenda2),  % breadth-first search
        extend_items_one(Agenda2,Table2,Goal,YN).
```

```
new_items(item(Goal1,[]),Table,Items) :-
        findall(item(Goal2,Goals),
                member(item(Goal2,[Goal1|Goals]),Table),
                Items).
new_items(item(Goal1,[Goal2|Goals1]),Table,Items) :-
        findall(item(Goal2,Goals2),(Goal2 ::- Goals2),Items1),
        findall(item(Goal1,Goals1),
                member(item(Goal2,[]),Table),
                Items2),
        append(Items1,Items2,Items).

memberv(Item,Table) :-
        member(Item2,Table),
        variant(Item,Item2).

variant(X,Y) :-
        \+ (\+ (numbervars(X,0,_),numbervars(Y,0,_),X == Y)).
```

The definition of the predicates `append`, `member`, `findall` and `numbervars` follows the standard conventions. `append` is a predicate for the concatenation of two lists. `member` is a predicate for membership of a list. When `findall(X,condition(X),Solutions)` has been proved, `Solutions` = $\{X \mid \mathit{condition}(X)\}$. `numbervars` replaces all variables in a term by special constants. This operation makes two terms identical when they are alphabetic variants. Two sample programs are (facts are rules with an empty body):

```
% Sample program PATH

path(X,Z) ::-
       [path(X,Y),
        edge(Y,Z)].
path(X,X) ::- [].

edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].
```

The predicates just given are repeated here with a little comment.

```
prove_all(Goal,Solutions) :-
       findall(item(Goal,Goals),(Goal ::- Goals),Agenda),
       extend_items_all(Agenda,[],Table),
       findall(Goal,member(item(Goal,[]),Table),Solutions).
```

The main goal is used to predict items. These items are put in the agenda. The prover is started with `extend_items_all`. When `extend_items_all` is finished we search in the table for all solutions.

```
extend_items_all([],Table,Table).
```

If the agenda is empty we are finished.

```
extend_items_all([Item|Agenda1],Table1,Table2) :-
        memberv(Item,Table1),
        extend_items_all(Agenda1,Table1,Table2).
```

If an item from the agenda is in the table, it can be discarded.

```
extend_items_all([Item|Agenda1],Table1,Table3) :-
        \+ memberv(Item,Table1),
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        append(Items,Agenda1,Agenda2), % depth-first
%       append(Agenda1,Items,Agenda2), % breadth-first
        extend_items_all(Agenda2,Table2,Table3).
```

If an item is not in the table as yet, it is added and new items are generated. These new items are put in front or behind the agenda, corresponding with depth-first and breadth-first behaviour respectively.

```
new_items(item(Goal1,[]),Table,Items) :-
        findall(item(Goal2,Goals),
                member(item(Goal2,[Goal1|Goals]),Table),
                Items).
```

If the item is a solution, it is combined with items in the table that wait for that solution (completion).

```
new_items(item(Goal1,[Goal2|Goals1]),Table,Items) :-
        findall(item(Goal2,Goals2),(Goal2 ::- Goals2),Items1),
        findall(item(Goal1,Goals1),
                member(item(Goal2,[]),Table),
                Items2),
        append(Items1,Items2,Items).
```

First we predict and then we complete.

```
memberv(Item,Table) :-
        member(Item2,Table),
        variant(Item,Item2).

variant(X,Y) :-
        \+ (\+ (numbervars(X,0,_),numbervars(Y,0,_),X == Y)).
```

Here we check whether there is an alphabetic variant in the table. Observe that two items never share any variables. *Variables are only shared within an item.*

The one-solution prover only differs from the all-solutions prover in the fact that the first stops when the first solution is found, whereas the latter goes on until the agenda is empty.

The one-solution interpreter has two advantages over the standard Prolog interpreter with a depth first strategy. The first is that for these simple programs the declarative meaning and the procedural meaning coincide: if the interpreter answers "yes" to a query then the query is indeed derivable from the facts. On the other hand, if an atom is derivable then the interpreter will answer "yes". Standard interpreters get in an infinite loop in our example program PATH. The second advantage is that "a problem is never solved twice". The reuse of results of subcomputations in Prolog interpreters is called *memoing* or *tabling*. The technique is also known in general as *dynamic programming*.

Memoing can save us a lot of time. Consider the program in Figure 2. It can be made arbitrarily long by adding `x4, x5, ...`.

```
s    :- x1a,c.
s    :- x1b.
x1a  :- x2a.
x1a  :- x2b.
x1b  :- x2a.
x1b  :- x2b.
x2a  :- x3a.
x2a  :- x3b.
x2b  :- x3a.
x2b  :- x3b.
x3a.
x3b.
```

Figure 2: Exponential versus quadratic.

When we use standard proof search the time needed to find the first proof is exponential in the size of the program. When we use Earley Deduction the time needed is quadratic in the length of the program (this will be explained later).

4 Space Complexity

Because it is much easier to reason about space complexity than about time complexity of the prover we gave in the previous section we start with a space complexity analysis. This analysis is also a good stepping stone towards the time complexity analysis in the next section. We will define three ways to look at decidability problems in Prolog theorem proving. The first is:

PROLOG THEOREM PROVING
INSTANCE: A query Q and a program P.
QUESTION: Is there a substitution f such that $f(Q)$ follows from P?

This problem is undecidable in general. It is semi-decidable: if the answer is yes we can give an algorithm that finds a proof in finitely many steps. An algorithm with this property is the one-solution prover. This prover searches its proofs under a breadth-first regime and is therefore guaranteed to stop if there exists a proof. The following problems are decidable for many programs P:

PROLOG THEOREM PROVING FOR PROGRAM P
INSTANCE: A query Q.
QUESTION: Is there a substitution f such that $f(Q)$ follows from P?

PROLOG THEOREM PROVING FOR A CLASS OF PROGRAMS
INSTANCE: A query Q and a program P, which is of the following form ....
QUESTION: Is there a substitution f such that $f(Q)$ follows from P?

In the rest of this paper we will only consider programs for which these problems are decidable. PROLOG THEOREM PROVING FOR PROGRAM P is interesting when the program is fixed. E.g. if our program sorts a list, and the list is in the query, then the program does not vary. The second kind of problems is relevant when the program can vary. Suppose we have a parsing problem where the input string is in the query and the lexicon is in the program, Both the input string and the lexicon can vary. If we want to express the complexity both in the length of the input string and in the length of the lexicon, we need PROLOG THEOREM PROVING FOR A CLASS OF PROGRAMS.

The space complexity both of PROLOG THEOREM PROVING FOR PROGRAM P and of PROLOG THEOREM PROVING FOR A CLASS OF PROGRAMS is the size of the table when the computation of the one-solution prover has finished.

This is not quite true because there can be many duplicates in the agenda. We have to modify the prover a bit. We have to make sure that items neither occur in the agenda nor in the table before we put them in the agenda. Then we know that the size of the agenda plus the size of the table is smaller than the size of the final table at any point in the computation. This can be implemented as follows in Prolog (for a depth-first behaviour the new items must be appended at the back of the agenda):

```
extend_items_all([],Table,Table).
extend_items_all([Item|Agenda1],Table1,Table3) :-
        Table2 = [Item|Table1],
        new_items(Item,Table1,Items),
        add_items_df(Items,Agenda1,Table2,Agenda2),
        extend_items_all(Agenda2,Table2,Table3).

add_items_df([],Ag,_,Ag).
add_items_df([H|T],Ag,Table,Ag2) :-
        memberv(H,Ag),
        add_items_df(T,Ag,Table,Ag2).
add_items_df([H|T],Ag,Table,Ag2) :-
        memberv(H,Table),
        add_items_df(T,Ag,Table,Ag2).
add_items_df([H|T],Ag,Table,Ag2) :-
        \+ memberv(H,Ag),
        \+ memberv(H,Table),
        add_items_df(T,[H|Ag],Table,Ag2).
```

Because we can not predict when the first solution is found by the one-solution prover, we will assume the worst scenario, where the computation ends because the agenda is empty. In fact we estimate the complexity of the one-solution prover and the all-solutions provers simultaneously.

In order to estimate the size of the final table we have to count

- the number of items in the table, and
- the length of the items.

We will index our clauses from now on. The function $l(i)$ denotes the number of atoms in the body of clause i. For every clause C_i :- $B_{i1}, B_{i2}, \ldots, B_{in}$ (i is the index, n replaces $l(i)$ for readability), the items in the table are of the following form:

$$\langle f(C_i), [f(B_{i1}), f(B_{i2}), \ldots, f(B_{in})]\rangle.$$
$$\langle \sigma_1(f(C_i)), [\sigma_1(f(B_{i2})), \ldots, \sigma_1(f(B_{in}))]\rangle.$$
$$\langle \sigma_2(\sigma_1(f(C_i))), [\sigma_2(\sigma_1(f(B_{i3}))), \ldots, \sigma_2(\sigma_1(f(B_{in})))]\rangle.$$
$$\vdots$$
$$\langle \sigma_n(\ldots(\sigma_1(f(C_i))), [\,]\rangle$$

If we want to know the number of items we have to estimate the number of possible substitutions f, $\sigma_1 \circ f$, $\sigma_2 \circ \sigma_1 \circ f$, ..., $\sigma_n \circ \ldots \circ f$, for every clause i in the program. In general the number of substitutions is infinite but we count here the number of substitutions for a given query and program under the proof procedure specified in the one-solution and all-solutions provers.

We first have to see which variables occur in the substitutions (or, for which variables the substitution is not the identity mapping). We will call these variables the *relevant variables*. Suppose we want to count the items of the following form:

$$\langle \sigma_j(\ldots\sigma_1(f(C_i))), [\sigma_j(\ldots\sigma_1(f(B_{i(j+1)}))), \ldots, \sigma_j(\ldots\sigma_1(f(B_{in})))]\rangle.$$

Then the following variables are *not* relevant:

- Variables that occur in $B_{i1}, \ldots, B_{ij}$, but neither in $B_{i(j+1)}, \ldots, B_{in}$ nor in C_i. These variables are not relevant, simply because they do not occur in the item.

- Variables that occur in $B_{i(j+1)}, \ldots, B_{in}$, but neither in $B_{i1}, \ldots, B_{ij}$ nor in C_i. These variables are not relevant, because they are uninstantiated.

The other variables are relevant. These can be divided in two groups:

- The variables in C_i. These are called the *head variables* $HV(i)$.

- Variables that occur both in $B_{i1}, \ldots, B_{ij}$ and in $B_{i(j+1)}, \ldots, B_{in}$ but not in C_i. These variables are called the *relevant body variables* $RV(i,j)$.

We introduce a notation for the number of possible substitutions for a set of variables.

For a given i, the function $\#(varset, j)$ ($varset$ is a set of variables, j is a number) returns the number of possible substitutions of the variables in $varset$ after proving B_{i1} through B_{ij}.

The number of items in the final table is:

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)} \#(HV(i) \cup RV(i,j), j)$$

We will apply this formula in the example program PATH.

```
% Sample program PATH

path(X,Z) ::-
        [path(X,Y),
         edge(Y,Z)].
path(W,W) ::- [].

edge(a,b) ::- [].
edge(b,c) ::- [].
edge(c,a) ::- [].
edge(c,d) ::- [].
```

The number of items is:
$\#(\{X,Z\} \cup \emptyset, 0) + \#(\{X,Z\} \cup \{Y\}, 1) + \#(\{X,Z\} \cup \emptyset, 2) +$
$\#(\{W\} \cup \emptyset, 0) +$
$\#(\emptyset \cup \emptyset, 0) +$
$\#(\emptyset \cup \emptyset, 0) +$
$\#(\emptyset \cup \emptyset, 0) +$
$\#(\emptyset \cup \emptyset, 0)$

We know that the variables in this programs can only be substituted by a vertex in the graph (a, b, c or d). We denote the number of vertices in the graph as $|V|$ and the number of edges as $|E|$, and fill in the formula:

$$|V|^2 + |V|^3 + |V|^2 + |V| + |E|$$

Because $|E| \leq |V|^2$, this equals $\mathcal{O}(|V|^3)$.

The space complexity does not depend on the number of items only, but also on the length of the items. We introduce the function $\#\#$ that does not count the number of substitutions, but the number of *symbols needed to write down* all substitutions. The number of instantiations and the length of the instantiations behave different if they are estimated for a set of variables. Suppose we have two variables, $A1$ and $A2$. The number of possible substitutions for $A1$ is n_1. The number of possible substitutions for $A2$ is n_2. The average length of the substitutions for $A1$ and $A2$ is l_1 and l_2 respectively. Then the number of possible substitutions for $\#(\{A,B\})$ is $n_1 \times n_2$. The average length of all substitutions is $l_1 + l_2$, and $\#(\{A,B\})$ is $(n_1 \times n_2) \times (l_1 + l_2)$. We have to *multiply* the possibilities and *add* the lengths.

We assume that the length of clauses in the program is bounded by a constant. The space complexity of PROLOG THEOREM PROVING FOR PROGRAM P is now:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)} \#\#(HV(i) \cup RV(i,j), j)$$

In the example, we assume that the number of symbols needed to write down a vertex is bounded by a constant. In that case, the space complexity of the problem is $\mathcal{O}(|V|^3)$.

5 Time Complexity

In the previous section we saw how the space complexity of a problem can be estimated. In this section we will consider the time complexity. Observe that the Earley interpreter is deterministic. Therefore it can be converted to a program in an imperative language in a straightforward way. We will assume that this has been done. We will describe the time complexity of PROLOG THEOREM PROVING FOR PROGRAM P in terms of this imperative prover.

We count the number of steps of the all-solutions prover because we can not predict when the first solution of a problem has been found. Therefore we assume that the algorithm terminates when the agenda is empty.

We know that all items in the table and the agenda are different. Therefore the procedure `extend_items_all` will be executed as many times as there are items in the final table: $\sum_{i=1}^{k}\sum_{j=0}^{l(i)}\#(HV(i)\cup RV(i,j),j)$. Within this procedure, we have to execute the procedures `new_items` and `add_items_df`. Within the procedure `new_items` we perform *prediction* and *completion*. We can divide the table in two parts. First, we have items of the form `item(Goal1,[Goal2|Goals])`. The number of items of this form is $\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#(HV(i)\cup RV(i,j),j)$. Secondly, we have items of the form `item(Goal,[])`. There are $\sum_{i=1}^{k}\#(HV(i),l(i))$ such items in the final table (observe that $RV(i,l(i))=\emptyset$). If we sum all completion steps, we see that every item of the form `item(Goal1,[Goal2|Goals])` is compared exactly once with every item of the form `item(Goal,[])` to check whether the second is a solution for the first. Therefore the total number of completion steps is

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#(HV(i)\cup RV(i,j),j)\times\sum_{i=1}^{k}\#(HV(i),l(i))$$

In a completion step, we have to unify two atoms. The time needed is the sum of the length of the two atoms. If the length of the atoms in the program, i.e. without variables substitutions, is bounded by a constant, then the total time needed for all completion steps is ($\#\#$ is the sum of the length of all possible substitutions):

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#\#(HV(i)\cup RV(i,j),j)\times\sum_{i=1}^{k}\#(HV(i),l(i)))\ +$$

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#(HV(i)\cup RV(i,j),j)\times\sum_{i=1}^{k}\#\#(HV(i),l(i)))$$

The number of prediction steps is estimated as follows. We do a prediction step once for every item of the form `item(Goal1,[Goal2|Goals])`, thus $\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#(HV(i)\cup RV(i,j),j)$ times. We have to compare `Goal2` with every clause in the program. Say the number of clauses in the program is $|P|$. Then we have to execute $\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#(HV(i)\cup RV(i,j),j)\times|P|$ unifications. Under the assumption that the length of the atoms in the program is bounded by a constant, the total time needed in the prediction steps amounts to $\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1}\#\#(HV(i)\cup RV(i,j),j)\times|P|$.

Remains to be counted the number of steps in `add_items_df`. Suppose we try to add new items immediately after every completion and prediction step. We organize our agenda and table in such a way that the time needed to look up and insert in the agenda and table is linear in the size of the item we want to insert. The length of the term that results after unification of two terms is at most the sum of the length of those two terms. After every completion step, which costs time proportional to the sum of the length of two items, we have to do a lookup and an insert which are also proportional in the sum of the length. Thus, we can multiply the number of completion steps by two. The same holds for the prediction steps. Efficient organisation of the agenda and the table implies that the movement from the agenda to the table gets a little more complicated. The time complexity of moving all items from the agenda to the table equals precisely the space complexity that we saw in the previous section. The total time needed for all operations together is:

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#(HV(i), l(i)) \ +$$

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#\#(HV(i), l(i)) \ +$$

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#(HV(i) \cup RV(i,j), j) \times |P| \ +$$

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)} \#\#(HV(i) \cup RV(i,j), j)$$

This can be simplified to:

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#(HV(i) \cup RV(i,j), j) \times (\sum_{i=1}^{k} \#(HV(i), l(i)) + |P|)) \ +$$

$$(\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#(HV(i) \cup RV(i,j), j) \times \sum_{i=1}^{k} \#\#(HV(i), l(i)))$$

We can use this formula to estimate the time complexity of the sample program PATH in the previous section: $\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#(HV(i) \cup RV(i,j), j) =$

$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#(HV(i) \cup RV(i,j), j) = \mathcal{O}(|V|^3)$

$\sum_{i=1}^{k} \#\#(HV(i), l(i)) = \sum_{i=1}^{k} \#(HV(i), l(i)) = \mathcal{O}(|V|^2 + |E|)$

$|P| = |E|$.

The time complexity is $\mathcal{O}(|V|^3 \times (|V|^2 + |E|)) = \mathcal{O}(|V|^5)$, because $|E| \leq |V|^2$.

The upper bound on the time complexity is high, merely because we use a very simple interpreter. It is possible to write an interpreter that is much more efficient. This has been done in Aarts (1995). It is e.g. possible to eliminate the unification in the completer step. This can be done as follows. Every item with a non-empty body is associated with two keys. The first key tells how the clause was "called". When we have proved the rest of the body, this key can be used to find all items that are waiting for this solution. The second key tells what items can be a solution for the first atom in the body. The keys are computed in the prediction step. With this usage of keys we can immediately see whether a given solution "fits" a waiting item. A disadvantage is that the number of items grows a little bit: we not only store the substitutions for the head variables after some part of the body has been proven, but we also store the substitutions when nothing has been proved yet (the "call").

With this more efficient interpreter we get the following formula for the time complexity. $BV(i,j)$ is the set of all variables of B_{ij}. The number $mc(j)$ denotes the number of clauses whose predicate name matches the predicate name of $B_{i(j+1)}$.

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} mc(j) \times \#\#([\langle HV(i),0\rangle, \langle HV(i) \cup RV(i,j), j\rangle, \langle BV(i,j+1), j+1\rangle)$$

For most Prolog programs we can reduce this formula. It is almost always the case that after unification with the head of a clause (at the moment of "calling"), the head variables are either ground, or not instantiated at all. *Definite programs* are programs in which all solutions (items with an empty body) are *ground*. If a program is definite and we have a query in which all argument positions are either ground or uninstantiated, we know that head variables are either ground, or not instantiated at all right after the "call". If a head variable X is not instantiated after the call, then $\#\#([\langle X,0\rangle]) = 1$. If a variable X is ground after the call, then $\#\#([\langle X,0\rangle]) = \#\#([\langle X,j\rangle])$ for all j. Thus, we can leave out all head variables $[HV(i),0]$ from the formula, if our program is definite and our query satisfies the condition just given. This leads to the formula:

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} mc(j) \times \#\#([\langle HV(i) \cup RV(i,j), j\rangle, \langle BV(i,j+1), j+1\rangle).$$

If we are not interested in the length of the program we can also leave out $mc(j)$ and get:

$$\sum_{i=1}^{k}\sum_{j=0}^{l(i)-1} \#\#([\langle HV(i) \cup RV(i,j), j\rangle, \langle BV(i,j+1), j+1\rangle).$$

6 Improved Earley Deduction

We can improve the Earley prover further as follows. Suppose we have a clause

```
s(A,E) :- det(A,B), n(B,C), v(C,D), np(D,E).
```

A simple observation is that the value of the variable E is unimportant until we have to prove `np(D,E)`. If we have two goals say `s(0,3)` and `s(0,6)` a lot of work will be done twice. Therefore we introduce the notion *relevant head variable*. Relevant head variables are variables occurring both in the head *and* before the point we have reached in the body. We will not describe in detail how this can be implemented in the interpreter.

The result of this improvement is that we only count the relevant head variables and not all head variables. Formally, we define the following set:

- $RHV(i,j)$ is the set of all the relevant head variables of clause i at position j, i.e. the set of variables occurring in B_{i1} through $B_{i(j-1)}$ *and* in the head.

$RV(i,j)$ is the set of all relevant body variables and $BV(i,j)$ is the set of all variables of Bij.

The complexity of a program is now:

$$\sum_{i=1}^{k} \sum_{j=0}^{l(i)-1} mc(j) \times \#\#([\langle RHV(i,j) \cup RV(i,j), j\rangle, \langle BV(i,j+1), j+1\rangle).$$

7 Further Research

As said in the introduction, the approach taken in this article is modest. We have proven an upper bound for the time complexity of Prolog programs in a pretty straightforward way. The bad thing is that the bound is often higher than one would desire. In order to get the right semantics we have to do the so-called *occur check* in every unification. Arguing against the occur check, Pereira (1993, p. 547) states:

> "it was felt that the cost of a procedure call in a reasonable programming language should not depend on the sizes of the actual parameters"

In the previous we saw that the complexity *does* depend on the size of the parameters. There are two sources for this. First we perform the occur check. Secondly, we make copies of the parameters when we store them in the tables and we have to compare parameters all the time. This comparison is often not necessary. E.g. if we have a DCG there will be many copies of tails of the input sentence. Instead of copying this list we would like to copy pointers to positions in this list. If we want to compare two terms in this case, we only have to see whether two pointers point to the same address. And we do not have to compare the two lists. In a practical system we could do the following. We first look whether two pointers point to the same address. If they do, then we are sure they are identical and we don't have to compare or copy anything. If they point to different addresses we have to compare them. It is still possible that they are the same.

```
firstpart([a,b]).
firstpart([a]).
secondpart([c]).
secondpart([b,c]).
pred1 :-
        firstpart(X),
        secondpart(Y),
        append(X,Y,Z),
        pred2(Z).
```

We have to compute twice whether `pred2([a,b,c])`, but the two terms `[a,b,c]` will be represented internally in a different way. Theoretically the "trick" does not help us much, but in practice it can be an improvement. In practice the size of the parameters will often be eliminated.

8 Existing Implementations of Earley Interpreters

Currently there are (at least) three implementations of the Earley interpreter. The first is a very experimental implementation by the author. This implementation can be found at `"http://www.fwi.uva.nl/~aarts/prolog.html"`. The interpreter has been written in Prolog and runs under Quintus, Sicstus and SWI Prolog. A nice feature, absent in other interpreters, is the possibility to inspect proof trees. This replaces the tracing facilities of standard Prolog in which inspects the proof search. The proof tree debugger stimulates a declarative programming style. The second implementation is SLG. This is also a Prolog in Prolog. It supports the third truth value for non-stratified programs. It has been developed by Chen and Warren. The third implementation is the successor of SLG, XSB. XSB has been developed at the University of New York at Stony Brook by Warren et al. This interpreters allows both Earley and standard deduction. The idea is that one can use the standard interpreter for simple, deterministic, predicates where tabling is a useless overhead. Complex non-deterministic predicates can be tabled. XSB has been written in C and should be competitive with other Prolog interpreters. The home page of the XSB Group is (at the moment of writing): `"http://www.cs.sunysb.edu/~sbprolog/"`.

References

Aarts, E. (1995). *Title tba.* PhD thesis, Research Institute for Language and Speech, Utrecht University.

Bratko, I. (1990). *PROLOG programming for artificial intelligence.* Wokingham, England: Addison-Wesley.

Lin, N.-W. (1993). *Automatic Complexity Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Arizona. Available as ftp://cs.arizona.edu/caslog/naiwei93.tar.Z.

Paterson, M. S., and Wegman, M. N. (1978). Linear unification. *Journal of Computer and System Sciences*, 16:158–167.

Pereira, F. (1993). Review of "The logic of typed feature structures" (Bob Carpenter). *Computational Linguistics*, 19(3):544–552.

Pereira, F., and Warren, D. H. (1983). Parsing as deduction. In *Proceedings of the 21st Ann. Meeting of the ACL.* Cambridge, Mass.: MIT Press.

Pereira, F. C., and Shieber, S. M. (1987). *Prolog and Natural Language Analysis*, vol. 10 of *CSLI Lecture Notes.* Stanford.

Tamaki, H., and Sato, T. (1986). OLDT resolution with tabulation. In *Proc. of 3rd Int. Conf. on Logic Programming*, 84–98. Berlin: Springer-Verlag.

Vieille, L. (1989). Recursive query processing: the power of logic. *Theoretical Computer Science*, 69:1–53.

Warren, D. H. D. (1975). Earley deduction, unpublished note.

Warren, D. S. (1992). Memoing for logic programs. *Communications of the ACM*, 35(3):94–111.