INDUCTIVE ASSERTION METHOD FOR LOGIC PROGRAMS

Włodzimierz Drabent† and Jan Małuszyński

Department of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden
computer mail: jmz@liuida.uucp

ABSTRACT

Certain properties of logic programs are inexpressible in terms of their declarative semantics. One example of such properties would be the actual form of procedure calls and successes which occur during computations of a program. They are often used by programmers in their informal reasoning. In this paper, the inductive assertion method for proving partial correctness of logic programs is introduced and proved sound. The method makes it possible to formulate and prove properties which are inexpressible in terms of the declarative semantics. An execution mechanism using the Prolog computation rule and arbitrary search strategy (eg. OR-parallelism or Prolog backtracking) is assumed. The method may be also used to specify the semantics of some extra-logical built-in procedures for which the declarative semantics is not applicable.

1. INTRODUCTION

One of the most attractive features of logic programs is their declarative semantics [Apt, van Emden][Lloyd]. It describes program meaning in terms of least Herbrand models and logical consequence. It states, informally speaking, that whatever is computed by a logic program is its logical consequence and whatever its logical consequence is may be computed (unless the interpreter gets into an infinite loop due to an imperfect search strategy). More precisely, if a goal $\leftarrow A$ succeeds with a substitution θ as an answer then $\forall A\theta$ is a logical consequence of the program. If $\forall A\theta$ is a logical consequence of the program then there exists a computation for $\leftarrow A$ giving an answer substitution σ which is more general then θ (there exists γ such that $\theta = \sigma\gamma$). The least Herbrand model of a program is equal to the set of all ground atomic formulas A for which there exists a successful computation for the goal $\leftarrow A$.

In most cases the declarative semantics is sufficient for dealing with logic programs. For instance it may form a basis for formal program synthesis [Hogger]. However, there are some important properties of logic programs which are inexpressible in terms of the declarative semantics. An example of such a property is the correctness of a mode declaration. It is also often the case that a Prolog procedure is written under the assumption that all its invocations are of a certain form (and does not work properly when called in another way). Consider, for example, the procedure

append(X–Y, Y–Z, X–Z).

which appends difference lists. When used with the two first arguments being variables it produces incorrect results (they are not difference lists). Another example is the procedure

---

† present address: Institute of Computer Science, Polish Academy of Sciences, P.O.Box 22, 00-901 Warszawa PKiN, Poland

This research has been partially supported by the National Swedish Board for Technical Development, projekt nr STUF 85-3166 and STU 86-3372.

permute:

```
permute( [ ], [ ] ).
permute( T, [E|P] ) :- remove( T, E, T1 ), permute( T1, P ).
remove( [H|T], H, T ).
remove( [H|T], E, [H|T1] ) :- remove( T, E, T1 ).
```

which loops (after producing one answer) when invoked with a variable as the first argument. In every day reasoning about logic programs it is often necessary to discuss the actual form of procedure calls and answers. Features of this kind will be called here run-time properties as they concern not only a program's answer but also its execution process. Of course they cannot be dealt with in terms of the declarative semantics.

The declarative semantics is also insufficient in that it cannot predict the actual form of an answer. Knowing that $\forall A\theta$ is a logical consequence of a program we cannot say which substitutions are the answers to the goal $\leftarrow A$ (we only know that there is *an* answer more general than θ). Consider two programs:

```
p(f(a)).          p(f(X)).
p(f(X)).          q(a).
q(a).
```

The declarative semantics of both programs is the same, but for a goal $\leftarrow p(Y)$ they give different sets of answers. Proving what the actual answers are is possible in our approach.

This paper describes an inductive assertion method for proving run time properties of logic programs. In this work we are inspired by the well-known results of [Floyd] and [Hoare] for imperative programs but, due to the rather different nature of logic programs, direct application of these results is not possible. Our assertions refer to the bindings of the arguments of a procedure at each possible call of this procedure and upon its completion. Our notion of correctness relies on such assertions; a program is correct iff the conditions expressed by the assertions of a procedure are satisfied whenever this procedure is called, and whenever it achieves a success. We deal only with partial correctness: a procedure may loop or fail but if the program is correct we still know that the arguments of every subsequent call have the properties expressed by the corresponding assertion. A similar problem is tackled in [Mellish] but the approach is different, based on abstract interpretation. An attempt to treat termination of logic programs in a formal way is presented in [Francez et al].

The rest of the paper is organized as follows. Section 2 introduces the notion of the asserted logic program. Section 3 contains an informal explanation of the method with some example proofs. Its purpose is to introduce intuitions facilitating understanding of Section 4 which presents the method in a formal way. This section also contains some comparisons with the abstract interpretation method. A proof of the main theorem of this section is presented separately in Section 5. Section 6 contains conclusions. This paper is a slightly modified version of [Drabent, Małuszyński].

2. LOGIC PROGRAMS WITH ASSERTIONS

In this section we introduce the notion of an asserted logic program. We assume familiarity with foundations of logic programming, as presented for instance in [Lloyd].

By a logic program we mean a set of Horn clauses of the form

$$a_0 :- a_1, \ldots, a_n. \qquad n \geq 0,$$

including a goal clause of the form

$$:- a_1, \ldots, a_n. \qquad n \geq 0$$

where each a_i is an atomic formula of the form $p(t_1,\ldots,t_m)$ $(m \geq 0)$ consisting of a m-ary predicate symbol p and terms $t_1,\ldots,t_m$. The terms have the standard syntax: they are either variables or are constructed from functors and variables (constants are zero-argument functors).

By an n-ary procedure q of a logic program we mean the set of all clauses of the program whose left-hand sides begin with the n-ary predicate letter q.

In the examples we will use the syntax of Edinburgh Prolog [Bowen et al] including the list notation (functors including constants beginning with a small letter, variables beginning with a capital letter, [] standing for the empty list, $[Head|Tail]$ for the list consisting of $Head$ and $Tail$, $[t_1,\ldots,t_n]$ for an n-element list).

In this paper the form of procedure calls and answers during execution of logic programs is treated formally in the framework of SLD-derivations. Nothing about search strategy is assumed; it may be, for instance, OR-parallelism or the backtracking of Prolog. But in order to be able to obtain nontrivial results, some limitations on the computation rule are needed. In this paper the Prolog computation rule is used (the leftmost atomic formula in a current goal is always selected).

Our intention is to describe the form of procedure arguments at every possible call and upon its completion, and to prove correctness of such descriptions. This resembles the idea of introducing assertions for imperative programs [Floyd, Hoare]. Assertions are logic formulas that characterize states (variable valuations) of imperative programs. These formulas are to be interpreted on the data domain referred to by the program. The assertions can be seen as a specification of a program. They facilitate understanding of programs and are used as a basis for program verification. For each statement S of a program two assertions, a precondition and a postcondition, are given. They describe, respectively, states before the execution of S and states after this execution.

Experience has shown that it is often more convenient to use *binary* assertions [Tarlecki] which involve two states. For example a postcondition for a statement may describe the relation between the input and output states of this statement (while a "normal", unary assertion describes a set of states). In our approach, in order to describe a logic program a unary precondition and a binary postcondition are associated with every predicate symbol p of the program. The precondition characterizes the arguments of every call of the procedure p, and the postcondition describes relations between these arguments and their final instances when a call succeeds. The pair of pre- and postcondition will be called here an *assertion.* A program with an assertion for every its predicate symbol is called an *asserted program.*

An asserted program is said to be *correct* iff, during its execution, for any procedure call the precondition of the procedure is satisfied, and upon a success of the call the postcondition is satisfied. Note that this is partial correctness. It does not say whether a success actually occurs. A formal definition of program correctness is given in Section 4.

Now we introduce a metalanguage for writing assertions for logic programs. The language of clauses (the logic programming language) will be referred to as the object language. The domain of interpretation for the metalanguage are (not necessarily ground) terms of the object language. This is because the metalanguage is intended to describe relations on (object language) terms. The functors and the predicate symbols of the metalanguage given in the definition below refer only to some basic operations and relations. We do not intend to give an exhaustive list of such symbols, nor to restrict ourselves to some minimal set.

DEFINITION 2.1 (of the metalanguage of assertions)

1. Variables:

   a. ${}^\bullet p_i$, $p_i^\bullet$ $(i = 1, \ldots, n)$ where p is an n-ary predicate of the object language.

   b. T, U, V, ...

Comment: ${}^\bullet p_i$ stands for the value of i-th argument of p at invocation of the procedure p. $p_i^\bullet$ stands for the value of this argument at success. T, U, V,... stand for any terms.

2. n-ary functors $(n \geq 0)$:

   a. n-ary functors of the object language.

   b. variables of the object language: $X, Y, Z, \ldots$ $(n = 0)$.

   c. ...

For the functors from the cases *a.* and *b.* the interpretation of a functor is the functor itself.

3. Terms: standard definition.

4. Predicate symbols: $=$, var, ground, $\prec$, $\cong$, ...

Interpretation:

$=$ – term equality,

var(T) iff T is an (object language) variable,

ground(T) iff T is an (object language) ground term,

$\mathtt{T} \prec \mathtt{U}$ iff T is a subterm of U,

$\mathtt{T} \cong \mathtt{U}$ iff the terms T and U are variants of each other (they differ at most in the names of their variables),

disconnected($\mathtt{V}_1, \ldots, \mathtt{V}_n$) iff no variable occurs in more than one of the terms $\mathtt{V}_1, \ldots, \mathtt{V}_n$,

subterm(T,U,I) iff $\mathtt{T} \prec \mathtt{U}$ and I is the corresponding selector (assuming any fixed way of assigning selectors to subterm occurrences).

5. Logical connectives and quantifiers: **true**, **false**, $\vee$, &, $\Rightarrow$, ...

6. Formulas: standard definition.

7. An *assertion* for the predicate p is an expression

$$p : \textbf{pre } F_1; \textbf{post } F_2$$

where F_1, F_2 are formulas which do not contain the variables ${}^\bullet q_i$, $q_i^\bullet$ for $q \neq p$ and $p_i^\bullet$ does not occur in F_1. F_1, F_2 are called the precondition and the postcondition for p. □

Sometimes it is necessary to add integer arithmetic to the metalanguage. In this case we add numbers, arithmetical functors and predicates with the obvious interpretation.

Let a be an (object language) atomic formula of the form $p(t_1, \ldots, t_n)$. We will often say "pre-(post-)condition for a" instead of "pre-(post-)condition for p".

DEFINITION 2.2

Let $a = p(t_1, \ldots, t_n)$.

1. *a satisfies its precondition* F_1 iff F_1 is true w.r.t. (any) interpretation in which the values of ${}^\bullet p_1, \ldots, {}^\bullet p_n$ are, respectively, $t_1, \ldots, t_n$.

2. Let σ be a substitution. $(a, a\sigma)$ *satisfies its postcondition* F_2 iff F_2 is true w.r.t. (any) interpretation in which the values of ${}^\bullet p_1, \ldots, {}^\bullet p_n$ are, respectively, $t_1, \ldots, t_n$ and the values of $p_1^\bullet, \ldots, p_n^\bullet$ are, respectively, $t_1\sigma, \ldots, t_n\sigma$. □

EXAMPLE 2.1

Let p be a three argument predicate symbol. This is an assertion for p:

p : **pre** $\mathrm{var}({}^\bullet p_2)$ & $\mathrm{var}({}^\bullet p_3)$ & ${}^\bullet p_2 \not\prec {}^\bullet p_1$ & ${}^\bullet p_3 \not\prec {}^\bullet p_1$;

post $p_2^\bullet = p_3^\bullet = [\,]$ $\vee$

$\neg\mathrm{ground}(p_2^\bullet)$ & $((\mathrm{var}(\mathtt{V}) \;\&\; \mathtt{V} \prec p_2^\bullet) \Rightarrow \mathtt{V} \prec p_3^\bullet)$

The precondition means that the second and the third arguments of p are variables which do not occur in the first argument. The postcondition means that either the second and the third arguments are empty lists or the second one is nonground and every variable occurring in it also occurs in the third argument. Note that this is actually a unary postcondition (since it is independent of the arguments of the call of p).

The atomic formula $p([1,2],X,Y)$ satisfies its precondition and $p([1,X],X,Y)$ does not. The postcondition is satisfied by $(p([1,2],X,Y),\ p([1,2],[\,],[\,]))$ and by $(p([1,2],X,Y),\ p([1,2],[V,Z],[pair(1,V),pair(2,Z)]))$.

The program below is a part of the program *serialise* [Bowen et al].

$$:\!- p(\mathtt{T},X,Y). \qquad \text{where } X \not\prec \mathtt{T}, Y \not\prec \mathtt{T} \tag{0}$$

$$p([\,],[\,],[\,]). \tag{1}$$

$$p([A|LA],\ [B|LB],\ [pair(A,B)|LC]) :\!- p(LA,LB,LC). \tag{2}$$

This program together with the assertion is an asserted program. (Note that formally it is a class of programs as a class of goal statements is specified. X and Y are object language variables while T stands for any term not containing these variables.) □

EXAMPLE 2.2 (of an asserted program)

The program from Example 2.1 (but without any conditions for T in (0)) and with the following assertion for p:

pre **true** ;
post $p_2^\bullet = p_3^\bullet = [\,] \ \vee$
$\mathrm{var}({}^\bullet p_2)\ \&\ \mathrm{var}({}^\bullet p_3)\ \&\ {}^\bullet p_2 \not\prec {}^\bullet p_1\ \&\ {}^\bullet p_3 \not\prec {}^\bullet p_1 \Longrightarrow$
$\neg\mathrm{ground}(p_2^\bullet)\ \&\ \ ((\mathrm{var}(\mathtt{V})\ \&\ \mathtt{V} \prec p_2^\bullet) \Rightarrow \mathtt{V} \prec p_3^\bullet)$ □

EXAMPLE 2.3 (of an asserted program)

The program from Example 2.1 with the following assertion for p:

pre $\mathrm{var}({}^\bullet p_2)\ \&\ \mathrm{var}({}^\bullet p_3)\ \&\ {}^\bullet p_2 \not\prec {}^\bullet p_1\ \&\ {}^\bullet p_3 \not\prec {}^\bullet p_1$;
post $p_2^\bullet = [\mathtt{V}_1,\ldots,\mathtt{V}_n],\ n \geq 0\ \ \&\ \ \forall_{i,j}\ \mathrm{var}(\mathtt{V}_i)\ \&\ (i \neq j \Rightarrow \mathtt{V}_i \neq \mathtt{V}_j)$.

The postcondition means that the second argument of p (at a success of p) is a list of different variables. □

3. INFORMAL INTRODUCTION TO THE PROOF METHOD

The section contains an informal and intuitive presentation of the content of Section 4. Some readers may prefer to skip it and refer directly to that section.

Let us discuss computations of a program P relating to its clause

$$a_0 :\!- a_1,\ldots,a_n. \tag{*}$$

The clause may be invoked only when a current subgoal, say b, is unifiable with a_0. As a result of the unification some of the variables occurring in a_0 will be instantiated to terms, not necessarily ground. Let V denote a variable occurring in (*) or in b. The value of V after the unification will be denoted by $\mathtt{V}_0$. The value of an unbound variable is the variable itself. So $\mathtt{V}_0 = \mathtt{V}$ for example if V does not occur in a_0.

Let a_1' be a_1 with every variable V substituted by $\mathtt{V}_0$. After the unification, a_1' becomes the current subgoal. Upon a success of a_1' the variable bindings are updated: the value of each V is denoted by $\mathtt{V}_1$, and a_1' with the new bindings is denoted by a_1''.

Note that the difference between $\mathtt{V}_0$ and $\mathtt{V}_1$ is due to binding some of the variables which occur both in $\mathtt{V}_0$ and in a_1'. The variables are being bound to terms which replace them in $\mathtt{V}_0$ giving $\mathtt{V}_1$. If there are no such variables then $\mathtt{V}_0 = \mathtt{V}_1$. Further, $\mathtt{V}_0$ and $\mathtt{V}_1$ may differ even if V does not occur in a_1.

EXAMPLE 3.1

1. Let $V_0 = V$, $a_1 = p(V,b) = a_1'$. Suppose that $a_1'' = p(f(c),b)$, then $V_1 = f(c)$.

2. Let $V_0 = f(X,Y)$, $U_0 = X$, $a_1 = q(U)$ then $a_1' = q(X)$. Suppose that $a_1'' = q(g(Z))$. Then $U_1 = g(Z)$, $V_1 = f(g(Z),Y)$. V does not occur in a_1 but $V_1 \neq V_0$ □

In the sequel of the computation, each a_i may become a current subgoal with current values of its variables. The current value of a variable V at this moment is denoted by $\mathtt{V}_{i-1}$ and a_i' is a_i with every variable V substituted by $\mathtt{V}_{i-1}$. Upon a success of a_i' the variable bindings are updated; a_i with these new bindings is denoted by a_i'' and the value of V at this moment is denoted by $\mathtt{V}_i$. The dependencies between $\mathtt{V}_{i-1}$ and $\mathtt{V}_i$ are of the same kind as discussed above for $i = 1$.

Now we are ready to present an informal definition of a *valuation sequence* for the clause (*) and the (sub-) goal b. This is a sequence $\rho_0, \ldots, \rho_n$ of substitutions such that there exists a program P (containing (*)) and a computation of P for which

$$\rho_i = \{\mathtt{V} \mapsto \mathtt{V}_i \mid \mathtt{V} \text{ is a variable occurring in (*) or } b\}.$$

Thus $a_i' = a_i\rho_{i-1}$ and $a_i'' = a_i\rho_i$. Note that the definition takes into account only what is implied by the very clause (*) and b. It does not depend on any other clauses. Every computation of any program where the subgoal b invokes the clause (*) has a corresponding valuation sequence for (*) and b. This is true also in the case of failures and backtracking. If a_i' fails then, in the corresponding valuation sequence, $\mathtt{V}_0, \ldots, \mathtt{V}_{i-1}$ are the values of V which actually occurred in the computation. Backtracking is understood here as an attempt to construct another computation. Note that a valuation sequence exists iff b is unifiable with a_0.

A formal definition of a valuation sequence is presented in the next section and is based on the following properties. Firstly, ρ_0 is a most general unifier of b and a_0. Then, the difference between ρ_{i-1} and ρ_i is such that there exists a substitution σ_i and $\rho_i = \rho_{i-1}\sigma_i$ (σ_i is actually a computed answer substitution for a_i'). Furthermore, σ_i may change only the values of those variables which occur in a_i' and it may not introduce variables which have already occurred in the computation but do not occur in a_i'.

EXAMPLE 3.2

Let $b = p(c,Z)$. Consider the clause

$$p(A,C) :- q(A,B), r(B,C), s.$$

One of the possible valuation sequences is

$A_0 = c$, $B_0 = B$, $C_0 = Z$,

$A_1 = c$, $B_1 = f(Y)$, $C_1 = Z$,

$A_2 = A_3 = c$, $B_2 = B_3 = f(d)$, $C_2 = C_3 = e$.

The reader may construct a corresponding program. For all valuation sequences $B_0 = B$, $A_0 = c = A_1 = A_2 = A_3$ and $B_2 = B_3$, $C_2 = C_3$. The other possible C_0 is $C_0 = C$. □

Let a_0'' be a_0 in which every variable V is substituted by $\mathtt{V}_n$. If P is a correct program, then $a_1', \ldots, a_n'$ must satisfy their preconditions and $(a_1', a_1''), \ldots, (a_n', a_n'')$ must satisfy their postconditions. The precondition for b and the postcondition for (b, a_0'') must hold as well.

The following verification criterion (cf. also Fig. 1) is proved in the next section and is a basis for our proof method. (For simplicity a goal clause $:- a_1, \ldots, a_n$ is represented as **goal** $:- a_1, \ldots, a_n$ where both the precondition and postcondition for **goal** are **true**).

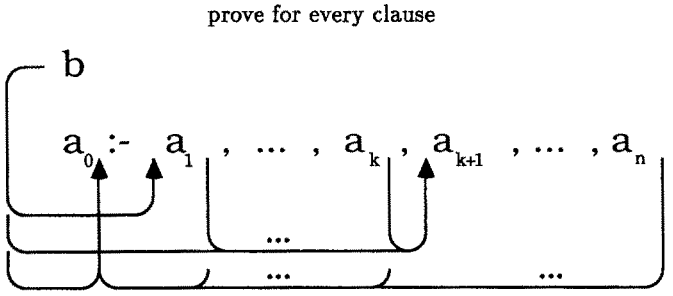

Figure 1. Verification condition, a diagram. Arrows stand for implications.

To prove that the program is correct, it is enough to prove for every clause $a_0 :- a_1, \ldots, a_n$ in the program ($n \geq 0$) that, for any goal b satisfying its precondition and any valuation sequence (for the clause and b),

1. the precondition for a'_1 holds,

2. for $k = 1, \ldots, n-1$, the precondition for a'_{k+1} is implied by the postconditions for $(a'_1, a''_1), \ldots, (a'_k, a''_k)$,

3. the postcondition for (b, a''_0) is implied by the postconditions for $(a'_1, a''_1), \ldots, (a'_n, a''_n)$.

An explanation for the above may be as follows. The correctness proof is divided into local proofs dealing with single clauses. For each clause $a_0 :- a_1, \ldots, a_n$ we can assume that the subgoal b invoking it satisfies its precondition. This should follow from the proofs related to the clauses involved in the computation leading to b as the current subgoal. But we have to prove that the precondition for a'_1 holds. Further, a'_1 may either fail or succeed giving a''_1. Since we already know that the precondition for a'_1 holds, it follows from the proofs for appropriate clauses that the postcondition for (a'_1, a''_1) holds. We can use this fact to prove the precondition for a'_2. Generally, to prove the precondition for a'_{k+1} it can be assumed that the postconditions for $(a'_1, a''_1), \ldots, (a'_k, a''_k)$ hold (because the preconditions for $a'_1, \ldots, a'_k$ are already proved). The same assumption, for $k = n$, can be used to prove the postcondition for (b, a''_0).

Note that for $n = 0$ it is enough to prove the postconditions for (b, a''_0) (the conditions 1. and 2. and the premises in 3. disappear). For $n = 1$ the case 2. disappears.

In our proofs we will use some abbreviations and notational conventions. Let (*) be the clause under consideration. When it does not lead to ambiguity, we will say that a precondition is satisfied by a_i (instead of the appropriate instance of a_i). The same for postconditions. If the predicate symbol of a_i is p, we will also say that the pre-(post-)condition for p is satisfied (or "... for p_i" if p occurs more than once in the clause). For example, in a proof for the clause $test(X) :- testa(cond1, X, Y), testb(Y), test(Y)$ we usually say "the postcondition for $testb$ is satisfied" instead of "the postcondition for (a'_2, a''_2) is satisfied" where a'_2 and a''_2 are appropriate instances of $testb(Y)$ (that means $a'_2 = testb(Y_1)$, $a''_2 = testb(Y_2)$).

By ${}^\bullet p_{i,j}$ and $p^\bullet_{i,j}$ we denote the value of the j-th argument of p_i at the moment of its invocation and its success respectively. The index i may be skipped when p occurs only once in the clause. So in the example above, ${}^\bullet test_{3,1} = Y_2$, $test^\bullet_{3,1} = Y_3$, ${}^\bullet testa_1 = testa^\bullet_1 = cond1$.

EXAMPLE 3.3 A correctness proof for the program from Example 2.1.

The proof for clauses (0) and (1) is immediate. Consider (2):

$$p([A|LA],\ [B|LB],\ [pair(A,B)|LC]) :- p(LA, LB, LC).$$

Let the head of (2) be unified with b satisfying its precondition; then $b = p(\mathtt{T}, \mathtt{X}, \mathtt{Y})$ where $\mathtt{X} \neq \mathtt{Y}$ (because of the occur check). Then B_0, LB_0, LC_0 are distinct variables and none of them occurs in LA_0. So the precondition for p_1 (strictly speaking, for $p(LA_0, LB_0, LC_0)$) holds.

It remains to prove that the postcondition for p_0 (that means for $(b,\ p([A_1|LA_1],\ [B_1|LB_1],\ [pair(A_1,B_1)|LC_1]))$) holds. $B_1 = B_0$ since B_0 does not occur in LA_0, LB_0, LC_0. So $\neg\text{ground}(p^{\bullet}_{0,2})$ (since $B_1 \prec p^{\bullet}_{0,2}$). Let V be a variable and $\mathtt{V} \prec p^{\bullet}_{0,2}$. Two cases are possible:

1. $\mathtt{V} = B_1 \prec p^{\bullet}_{0,3}$,
2. $\mathtt{V} \prec LB_1$ and from the postcondition for p_1 we obtain $\mathtt{V} \prec LC_1 \prec p^{\bullet}_{0,3}$. Q.E.D.

EXAMPLE 3.4 A correctness proof for the program from Example 2.3.

The proof for (0) and (1) is trivial. The clause (2) and the precondition for p are the same as in Example 3.3. As we have already proved, the precondition for p_1 holds and $B_1 = B_0$ & $\text{var}(B_1)$. From the postcondition for p_1

$LB_1 = [\mathtt{V}_1,\ldots,\mathtt{V}_n]$, $n \geq 0$ & $\forall_{i,j}\ \text{var}(\mathtt{V}_i)$ & $(i \neq j \Rightarrow \mathtt{V}_i \neq \mathtt{V}_j)$.

So $p^{\bullet}_{0,2} = [B_1, \mathtt{V}_1,\ldots,\mathtt{V}_n]$. We have also $\forall_i\ B_1 \neq \mathtt{V}_i$ because B_1 does not occur in the invocation of p_1 (see also section 4, the definition of valuation sequence, condition 4). Hence the postcondition for p_0 holds. Q.E.D.

4. THE METHOD

The main part of this section is a definition of program correctness and the verification theorem. These are preceded by a few necessary definitions and followed by examples. The section concludes with some comparisons between our method and abstract interpretation.

Let t be a term and $\theta = \{V_1 \mapsto t_1,\ldots,V_n \mapsto t_n\}$ a substitution. The following notation will be used:

$\text{variables}(t)$ is the set of (object language) variables occurring in t,
$\text{variables}(t_1,\ldots,t_n) = \text{variables}(t_1) \cup \ldots \cup \text{variables}(t_n)$,
$\text{dom}(\theta) = \{V_1,\ldots,V_n\}$,
$\text{variables}(\theta) = \text{variables}(t_1,\ldots,t_n)$. □

We use the traditional definition of SLD-derivation as presented in [Lloyd] restricting it to the fixed computation rule of Prolog. We must make explicit some assumptions which are stated there, but not precisely enough. For a most general unifier (mgu) θ of t_1 and t_2 we require that it does not introduce new variables:

$\text{variables}(\theta) \subseteq \text{variables}(t_1) \cup \text{variables}(t_2)$.

Note that θ does not use unnecessary variables:

$\text{dom}(\theta) \subseteq \text{variables}(t_1) \cup \text{variables}(t_2)$.

For an SLD-derivation we require that variables are standardized apart. That is, if

$G_0, G_1,\ldots; C_1, C_2,\ldots; \theta_1, \theta_2,\ldots$ is an SLD-derivation then for every $i < j$
$\text{variables}(C_i) \cap \text{variables}(C_j) = \emptyset$ and
$\text{variables}(G_i) \cap \text{variables}(C_j) = \emptyset$

(where $G_0, G_1,\ldots$ is the goal sequence, $C_1, C_2,\ldots$ is the clause variant sequence and $\theta_1, \theta_2,\ldots$ is the unification sequence of the derivation; the sequences may be finite or infinite).

DEFINITION 4.1

An asserted program P is *correct* iff for every SLD-derivation of P where $G_0, G_1,\ldots$ is the sequence of goal clauses and $\theta_1, \theta_2,\ldots$ is the sequence of substitutions and for every i if

$$G_i = :\!-a_1,\ldots,a_m, \qquad m \geq 0$$

then

1. a_1 satisfies its precondition,
2. if there exists $j > i$ such that

$$G_j = :\!-(a_2,\ldots,a_m)\theta_{i+1}\ldots\theta_j$$

then $(a_1, a_1\theta_{i+1}\ldots\theta_j)$ satisfies its postcondition for the least such j. □

Informally, a_i is a procedure call, $\theta_{i+1} \ldots \theta_j$ the corresponding computed answer substitution, $a_1\theta_{i+1} \ldots \theta_j$ the instantiation of a_1 at the moment of its success. The part of the SLD-derivation between i and j is, intuitively, the computation corresponding to procedure call a_1.

To facilitate formulation of the main theorem we introduce the notion of a valuation sequence.

DEFINITION 4.2

A sequence of substitutions $\rho_0, \ldots, \rho_n$ $(n \geq 0)$ is a *valuation sequence* for a clause $a_0 :- a_1, \ldots, a_n$ and for an atomic formula (a goal) b iff

0. $\text{variables}(b) \cap \text{variables}(a_0, a_1, \ldots, a_n) = \emptyset$
1. ρ_0 is an mgu of b and a_0

and there exist $\sigma_1, \ldots, \sigma_n$ (called an *answer sequence*) such that for $i = 1, \ldots, n$

2. $\rho_i = \rho_{i-1}\sigma_i$
3. $\text{dom}(\sigma_i) \subseteq \text{variables}(a_i\rho_{i-1})$
4. $\text{variables}(\sigma_i) \cap \text{variables}((a_0 :- a_1, \ldots, a_n)\rho_{i-1}) \subseteq \text{variables}(a_i\rho_{i-1})$. □

ρ_i can be understood as a valuation of clause variables upon a success of $a_i\rho_{i-1}$ (provided it succeeded). σ_i is the corresponding computed answer substitution. Using the notation from the previous section, $\mathtt{V}\rho_i = \mathtt{V}_i$ for any variable $\mathtt{V}$ occurring in the clause.

The theorem below is the main result of this paper and the basis of our proof method. In the theorem we assume that $a_0 = \mathbf{goal}$ for a goal clause where **goal** is a special predicate symbol which does not occur elsewhere. The assertion for **goal** is **pre true; post true.**

THEOREM 4.1 (verification condition)

Let P be an asserted program. A sufficient condition for P to be correct is:

for every $a_0 :- a_1, \ldots, a_n$ being a clause of P $(n \geq 0)$,
for every b which satisfies its precondition,
for every their valuation sequence $\rho_0, \ldots, \rho_n$

1. the precondition for $a_1\rho_0$ is satisfied,
2. for every $k = 1, \ldots, n-1$, if $(a_1\rho_0, a_1\rho_1), \ldots, (a_k\rho_{k-1}, a_k\rho_k)$ satisfy their postconditions then the precondition for $a_{k+1}\rho_k$ is satisfied,
3. if $(a_1\rho_0, a_1\rho_1), \ldots, (a_n\rho_{n-1}, a_n\rho_n)$ satisfy their postconditions then the postcondition for $(b, a_0\rho_n)$ is satisfied. □

Note that for a unary clause $(n = 0)$ the conditions 1., 2., 3. above reduce to

3. the postcondition for $(b, a_0\rho_0)$ is satisfied.

For $n = 1$ they reduce to

1. the precondition for $a_1\rho_0$ is satisfied,
3. if $(a_1\rho_0, a_1\rho_1)$ satisfy its postcondition then the postcondition for $(b, a_0\rho_1)$ is satisfied.

The verification condition is expressed in semantic terms. While proving implications 2. and 3. one has to refer to properties of substitution composition, substitution application and unification. An interesting problem is finding a set of proof rules which would correspond to theorem 4.1 and would allow to perform proofs in a syntactic way, like in the axiomatic semantics. This could make possible automatization of the method.

Two example proofs of program correctness are given at the end of the previous section. Here we present another two examples relating to mode declarations.

EXAMPLE 4.1

Consider the following program

$$:- q(\mathtt{T}). \tag{0}$$
$$q(L) :- p(L,M,N),\ s(N,L1,L2). \tag{1}$$
$$p([\,],[\,],[\,]). \tag{2}$$
$$p([A|LA],\ [B|LB],\ [pair(A,B)|LC]) :- p(LA,LB,LC). \tag{3}$$
$$s([\,],[\,],[\,]). \tag{4}$$
$$s([X|L],[X|L1],L2) :- s(L,L1,L2). \tag{5}$$
$$s([X|L],L1,[X|L2]) :- s(L,L1,L2). \tag{6}$$

(where the procedure p is the same as in the previous examples) with the assertions

q : **pre true; post true**

p : **pre true; post** $p_3^\bullet = [\mathtt{T}_1,\ldots,\mathtt{T}_n],\ n \geq 0\ \ \&\ \ \forall_i \neg\mathrm{var}(\mathtt{T}_i)$

s : **pre** ${}^\bullet s_1 = [\mathtt{T}_1,\ldots,\mathtt{T}_n],\ n \geq 0\ \ \&\ \ \forall_i \neg\mathrm{var}(\mathtt{T}_i)\ \ \&\ \ \mathrm{var}({}^\bullet s_2)\ \ \&\ \ \mathrm{var}({}^\bullet s_3)$; **post true**

As the correctness proof for the program is easy, we present here proofs for clauses (1) and (5) only.

A proof for (1): Let the head of (1) be unified with b satisfying its precondition. As the precondition is **true**, $b = q(\mathtt{S})$ (where $\mathtt{S}$ is any term) and $\rho_0 = \{L \mapsto \mathtt{S}\}$ or, if $\mathtt{S}$ is a variable, $\rho_0 = \{\mathtt{S} \mapsto L\}$. Let $a_1 = p(L,M,N)$ and $a_2 = s(N,L1,L2)$. Then $a_1\rho_0$ satisfies its precondition. Assume that $(a_1\rho_0, a_1\rho_1)$ satisfies its postcondition. This means that $N\rho_1 = [\mathtt{T}_1,\ldots,\mathtt{T}_n]$, $n \geq 0\ \ \&\ \ \forall_i \neg\mathrm{var}(\mathtt{T}_i)$ and the precondition for $a_2\rho_1 = s((N\rho_1),L1,L2)$ holds. This completes the proof for (1) since the postcondition for q is **true**.

A proof for (5): Let $b = s([\mathtt{T}_1,\ldots,\mathtt{T}_n],\mathtt{U},\mathtt{V})$ satisfies its precondition (this means $n \geq 0$, $\mathtt{U},\mathtt{V}$ are variables, $\mathtt{T}_1,\ldots,\mathtt{T}_n$ are not variables). Let b be unified with the head of (5) by mgu ρ_0. Then $n \geq 1$,

$$b\rho_0 = s([X|L],[X|L1],L2)\rho_0 = s([\mathtt{T}_1|[\mathtt{T}_2,\ldots,\mathtt{T}_n]],[\mathtt{T}_1|\mathtt{W}],\mathtt{X})$$

(where $\mathtt{W},\mathtt{X}$ are variables) and

$$s(L,L1,L2)\rho_0 = s([\mathtt{T}_2,\ldots,\mathtt{T}_n],\mathtt{W},\mathtt{X})$$

which satisfies its precondition. This completes the proof for (5) since the postcondition for s is **true**.

From the precondition for s it follows that the procedure s may be given a mode declaration $s(+,-,-)$ (since at every call of s the first argument is not a variable and the remaining arguments are variables). □

EXAMPLE 4.2

Consider the program fragment

$$p :- q(f(a),X),\ r(X). \tag{1}$$
$$s(Y) :- q(Y,X),\ t(X). \tag{2}$$
$$q(f(X),X). \tag{3}$$

with the assertions

p : **pre true; post true**

q : **pre true; post** $\mathrm{ground}({}^\bullet q_1) \Rightarrow \mathrm{ground}(q_2^\bullet)$

r : **pre** $\mathrm{ground}({}^\bullet r_1)$; **post true**

Let all the remaining assertions be **pre true; post true.** It is easy to prove that this asserted program is correct (under the assumption that the procedure q consists only of (3) and that the only invocation of r occurs in (1)). So the procedure r may be given a mode declaration $r(+)$. □

Neither of the mode declarations that are shown to be correct in these examples can be found using the abstract interpretation method of [Mellish]. In Example 4.1 this is because of too restricted domains of abstract interpretation. To find the mode declaration for s it is necessary to know that p_3^* is a list of non-variable elements, but the abstract interpreter supports no description between "ground term" and "term whose arguments are variables". (Actually, this shows why the abstract interpreter is not able to find an adequate mode declaration for the procedure *split* in the program *serialize* [Bowen et al], since the procedure s is a simplified version of *split*). To find the mode declaration from Example 4.2 it is necessary to treat the calls of q in (1) and (2) in a different way. This is possible in our approach (implications in a binary postcondition can be used for this purpose) but impossible in the abstract interpretation method mentioned above.

This weakness of abstract interpretation is due to its automaticity: the same apparatus is applied to every program while a proof method like ours can use assertions tailored to the problem on hand (cf. Examples 2.1, 2.2, 2,3, 4.1 where four distinct assertions are given to the same procedure).

5. PROOF OF THE VERIFICATION THEOREM

This section proves the soundness of our method. To facilitate the proof we introduce some definitions. Let $G_0, G_1, \ldots; C_1, C_2, \ldots; \theta_1, \theta_2, \ldots$ be an SLD-derivation (the reader is referred to [Lloyd] for standard definitions and theorems).

DEFINITION

A k,l-*subrefutation* (of this derivation) is $G_{k-1}, \ldots, G_l; C_k, \ldots, C_l;\ \theta_k, \ldots, \theta_l$ such that
$G_{k-1} = :\!-b, b_1, \ldots, b_m, \qquad m \geq 0$
$G_l = :\!-(b_1, \ldots, b_m)\theta_k \ldots \theta_l$
and l is the least such number. □

DEFINITION

A k,j-*subderivation* (of this derivation) is $G_{k-1}, \ldots, G_j; C_k, \ldots, C_j;\ \theta_k, \ldots, \theta_j$ such that
$G_{k-1} = :\!-b, b_1, \ldots, b_m, \qquad m \geq 0$
and for $k \leq i \leq j$ G_i is not of the form $:\!-(b_1, \ldots, b_m)\theta_k \ldots \theta_i$. □

A subrefutation beginning with $:\!-b, \ldots$ is a fragment of an SLD-derivation related to a successful procedure call b. A subderivation beginning with the same goal may be treated as a not yet completed computation associated with b.

The sufficient condition from the Theorem 4.1 will often be referred to as (SC).

LEMMA 5.1

Let $G_{k-1}, \ldots, G_l; C_k, \ldots, C_l; \theta_k, \ldots, \theta_l$ be a subrefutation of an SLD-derivation of a program P for which (SC) is satisfied. Let $k \leq i \leq l$ and
$G_{k-1} = :\!-b, b_1, \ldots, b_m,$
$\sigma_i = \theta_k, \ldots, \theta_i,$
$G_i = :\!-(A_i, b_1, \ldots, b_m)\sigma_i$, where A_i is a sequence of atomic formulas.
Then
$\mathrm{dom}(\sigma_i) \subseteq \mathrm{variables}(b, C_k, \ldots, C_i)$,
$\mathrm{variables}(\sigma_i) \subseteq \mathrm{variables}(b, C_k, \ldots, C_i)$ and
$\mathrm{variables}(A_i\sigma_i) \subseteq \mathrm{variables}(b, C_k, \ldots, C_i)$. □

COROLLARY

Let $G_0, G_1, \ldots; C_1, C_2, \ldots; \theta_1, \theta_2, \ldots$ be an SLD-derivation of a program for which (SC) is satisfied. Let there exist a k,l-subrefutation of the derivation. Let $G_{k-1} = :-b, b_1, \ldots, b_m$. Then

$$G_{k-1}\theta_k \ldots \theta_l = G_{k-1}\sigma$$

where

$$\sigma = \theta_k \ldots \theta_l | \text{variables}(b) \quad \text{(and } | \text{ is defined by } \theta|X = \{V \mapsto t \in \theta \mid V \in X\}\text{)}.$$

More generally, for every $s \leq k$

$$G_{s-1}\theta_s \ldots \theta_l = G_{s-1}\theta_s \ldots \theta_{k-1}\sigma \quad \square$$

LEMMA 5.2

Let $G_0, G_1, \ldots; C_1, C_2, \ldots; \theta_1, \theta_2, \ldots$ be an SLD-derivation of a program for which (SC) is satisfied. Let a k,l-subrefutation of the derivation exist. Let $G_{k-1} = :-b, b_1, \ldots, b_m$ where b satisfies its precondition. Then $(b, b\theta_k \ldots \theta_l)$ satisfies its postcondition. $\square$

PROOF by induction on l.

Let the premises of the lemma hold.

$l = k$:

Let G_k be derived from G_{k-1} and a unary clause a_0 using an mgu θ_k. Then from (SC) follows the postcondition for $(b, a_0\theta_k)$.

$l > k$:

Let the lemma hold for every number less than l. Then

$$G_k \;=\; :-(a_1, \ldots, a_n, b_1, \ldots, b_m)\theta_k$$

is derived from G_{k-1} and a clause $C_k = a_0 :- a_1, \ldots, a_n$, $n > 0$. The substitution θ_k is an mgu of b and a_0.

There exist $r_0, \ldots, r_n$ such that $r_0 = k$, $r_n = l$ and, for $i = 1, \ldots, n$,

$$G_{r_i} = :-(a_{i+1}, \ldots, a_n, b_1, \ldots, b_m)\theta_k \ldots \theta_{r_i},$$

the derivation has a $(r_{i-1}+1), r_i$-subrefutation, and r_i is the least index for which it holds. The $(r_{i-1}+1), r_i$-subrefutation can be understood as a successful execution of the procedure call $a_i\theta_k \ldots \theta_{r_{i-1}}$.

Let $\rho_0 = \theta_k$ and for $i = 1, \ldots, n$

$$\sigma_i = \theta_{r_{i-1}+1} \ldots \theta_{r_i} | \text{variables}(a_i\theta_k \ldots \theta_{r_{i-1}}),$$

and $\rho_i = \rho_{i-1}\sigma_i$ (σ_i may be treated as a computed answer substitution for goal $a_i\theta_k \ldots \theta_{r_{i-1}}$). We want to prove that $\rho_0, \ldots, \rho_n$ is a valuation sequence for b and C_k. It remains to show that conditions 3 and 4 of Definition 4.2 hold.

Let $G = G_k$ or $G = G_{k-1}\theta_k$. From the Corollary it follows that for $i = 0, \ldots, n-1$ if

$$G\theta_{k+1} \ldots \theta_{r_i} = G\sigma_1 \ldots \sigma_i$$

then

$$G\theta_{k+1} \ldots \theta_{r_{i+1}} = G\sigma_1 \ldots \sigma_{i+1}.$$

By induction $G\theta_{k+1} \ldots \theta_{r_i} = G\sigma_1 \ldots \sigma_i$ for $i = 0, \ldots, n$. Hence

$$b\rho_n = b\theta_k \ldots \theta_l,$$

$$a_i\rho_{i-1} = a_i\theta_k \ldots \theta_{r_{i-1}} \qquad (*)$$

(and $\text{dom}(\sigma_i) \subseteq \text{variables}(a_i\rho_{i-1})$ which is condition 3 of Definition 4.2),

$$a_i\rho_i = a_i\theta_k \ldots \theta_{r_i}.$$

By Lemma 5.1 applied to the $(r_{i-1}+1), r_i$-subrefutation (where $G_{r_{i-1}} = :-a_i\rho_{i-1}, \ldots$ by (*)) $\text{variables}(\sigma_i) \subseteq \text{variables}(\theta_{r_{i-1}+1} \ldots \theta_{r_i}) \subseteq \text{variables}(a_i\rho_{i-1}, C_{r_{i-1}+1}, \ldots, C_{r_i})$. Hence $\text{variables}(\sigma_i) \cap \text{variables}((a_0, \ldots, a_n)\rho_{i-1}) \subseteq \text{variables}(a_i\rho_{i-1})$ (since variables in the derivation

are standardized apart and variables$((a_0,\ldots,a_n)\rho_{i-1}) \cap$ variables$(C_j) = \emptyset$ for $j > r_{i-1}$). We have proved that $\rho_0,\ldots,\rho_n$ is a valuation sequence for b and C_k.

Now, by (SC1), the precondition for $a_1\rho_0$ is satisfied.

If the precondition for $a_i\rho_{i-1}$ is satisfied then the postcondition for $(a_i\rho_{i-1}, a_i\rho_i)$ is satisfied (for every $i = 1,\ldots,n$, by the inductive assumption).

The preconditions for $a_2\rho_1,\ldots,a_n\rho_{n-1}$ hold (by (SC2)).

The postcondition for $(b, b\rho_n)$ holds (by (SC3)).

But $b\rho_n = b\theta_k\ldots\theta_l$ which completes the proof. □

LEMMA 5.3

Let $G_0, G_1,\ldots; C_1, C_2,\ldots; \theta_1, \theta_2,\ldots$ be an SLD-derivation of a program for which (SC) is satisfied. Then for every s the first atomic formula of G_s satisfies its precondition. □

PROOF by induction on s.

If $s = 0$ then the thesis follows immediately from (SC1). Let the lemma hold for every number less than s. Two cases are possible.

1.

$$\begin{aligned} G_s &= :-(a_1,\ldots,a_n,b_1,\ldots,b_m)\theta_s, \qquad n > 0 \\ G_{s-1} &= :-b,b_1,\ldots,b_m \end{aligned}$$

The precondition for b is satisfied and G_s is derived from G_{s-1} and a clause $a_0{:-}a_1,\ldots,a_n$. θ_s is an mgu of b and a_0. From (SC1) it follows that the precondition for $a_1\theta_s$ is satisfied.

2. (n in the previous case is 0)

There exists $k < s$ $(k \geq 0)$ such that

$$\begin{aligned} G_s &= :-(b_1,\ldots,b_m)\theta_k\ldots\theta_s \\ G_{s-1} &= :-(b_0,b_1,\ldots,b_m)\theta_k\ldots\theta_{s-1} \\ G_k &= :-(a_1,\ldots,a_t,b_1,\ldots,b_u,\ldots b_m)\theta_k \\ G_{k-1} &= :-b,b_{u+1},\ldots,b_m. \end{aligned}$$

Let k be the greatest such number (when $k = 0$ then let $G_{-1} = :-\mathbf{goal}$, $\theta_0 = \epsilon$ and C_0 be the goal clause $\mathbf{goal}{:-}\ldots$). Repeating the construction from the proof of Lemma 5.2 using $a_1,\ldots,a_t,b_1,\ldots,b_u$ instead of $a_1,\ldots,a_n$ and introducing r_v only for $v \leq t$ $(r_0 = k,\ r_t = s)$ we prove that the precondition for b_1 is satisfied. The evaluation sequence under consideration (for b and C_k) is $\rho_0,\ldots,\rho_{t+u}$ where $\rho_i = \rho_{i-1}\sigma_i$. σ_i is as in the previous proof for $i = 1,\ldots,t$. For $i = t+1,\ldots,t+u$, $\sigma_i = \epsilon$. We omit details of the proof. □

Theorem 4.1 follows immediately from lemmas 5.3 and 5.2.

6.CONCLUSIONS

In this paper, the inductive assertion method for logic programs was introduced and proved sound. The metalanguage of assertions was defined. The assertions can describe properties that are inexpressible in terms of the declarative semantics. The verification theorem makes it possible to prove the partial correctness of programs with respect to their assertions.

We think that the ability of stating and proving assertions is important for the following reasons:

1. Assertions may improve the legibility of some logic programs. They may be treated as formalized comments specifying the actual form of procedure calls and successes.

2. Prolog programmers quite often reason about their programs in terms of execution (this is reflected by comments, mode declarations, etc.). By introducing assertions one makes explicit some facts upon which this reasoning is based.

3. Intuitive principles of reasoning about logic programs can be formulated as a systematic method for proving the correctness of a logic program.

4. The declarative semantics gives no formal explanation of the concept of the "logical variable" essential in many applications. The introduction of a metalanguage that refers to non-ground terms should make it possible to handle this concept in a more rigorous way.

5. It may be conceivable to use a metalanguage similar to the one presented here in logic programming systems. A debugging tool might use assertions to perform additional checking. A compiler might use them to guide optimizations.

Our approach can easily be extended to deal with some extra-logical built-in procedures. It can provide their formal semantics and also the absence of some run-time errors can be proved. The declarative semantics is inapplicable to this kind of procedures.

EXAMPLE Axiomatic semantics of the Prolog [Bowen et al] built-in procedure *var*
The meaning of the procedure may be described by the assertion
var : **pre true** ;
post var(•var_1) & •$var_1 = var_1^\bullet$. □

EXAMPLE Correctness of use of the Prolog built-in procedure *is*
Consider the assertion
is : **pre** intexpr(•is_2) ;
post true
where intexpr(T) iff T is an expression built out of integers and arithmetical functors. If an asserted program with the above assertion is correct then no run-time error connected with wrong arguments of *is* occurs. □

Our method is valid for the Prolog computation rule and for every search strategy (thus including OR-parallelism). It is also valid for Prolog programs containing the cut and negation-as-failure (although it is not able to exploit specific properties of the cut and *not*, cf. the assertion for *not*: **pre true**; **post** $not_1^\bullet = \bullet not_1$).

ACKNOWLEDGEMENTS

Thanks are due to Henryk Jan Komorowski for his critical comments. Ivan Rankin helped to improve the English of the previous version of this paper.

REFERENCES

[Apt, van Emden] Apt, K.R. and van Emden, M.H., "Contributions to the Theory of Logic Programming", J.ACM. 29, 3 (July 1982), 841-862

[Bowen et al] D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira and D.H.D. Warren, "Prolog-20 user's manual", 1984

[Drabent, Małuszyński] W.Drabent and J. Małuszyński , Proving runtime properties of logic programs, Research Report LITH-IDA-R-86-23, Linköping University, July 1986

[Floyd] Floyd, R.W., "Assigning Meanings to Programs", Proc.Symp.Appl.Math., Vol. 19: Mathematical Aspects of Computer Science (J.T.Schwartz, ed.), pp. 19–32, American Mathematical Society, Providence, Rhode Island, 1967

[Francez et al] Francez,N., Grumberg,O., Katz,S., Pnuelli,A., "Proving Termination of Prolog Programs" in "Logics of Programs. Proceedings, 1985", ed. by R.Parikh, Springer Lecture Notes in Computer Science 193, 89–105

[Hoare] Hoare,C.A.R. "An Axiomatic Basis for Computer Programming", Comm. ACM 12, 10 (Oct. 1969), 576–580,583

[Hogger] Hogger,C.J. "Derivation of Logic Programs", J.ACM 28, 2 (April 1981), 372–392

[Lloyd] Lloyd,J.W. "Foundations of Logic Programming", Springer-Verlag 1984

[Mellish] Mellish,C.S. "Abstract Interpretation of Prolog Programs", Third International Conference on Logic Programming, London, July 1986 and "The Automatic Generation of Mode Declarations for Prolog Programs", DAI Reaserch Paper 163, Dept of Artificial Intelligence, University of Edinburgh, 1981

[Tarlecki] Tarlecki,A., "A Language of Specified Programs", Science of Computer Programming 5 (1985) 59–81