IT IS DECLARATIVE

ON REASONING ABOUT LOGIC PROGRAMS

Włodzimierz Drabent[1]

December 12, 1997

Abstract

We advocate using the declarative reading in proving partial correctness of logic programs, when the properties of interest are declarative. Some recently published work presents unnecessarily complicated methods for proving such properties. These approaches refer to the operational semantics, as they consider calls and successes of the predicates of the program during LD-resolution. We show that this is an unnecessary complication and that a straightforward proof method is simpler and more general. Our approach is based solely on the property that "whatever is computed is a logical consequence of the program". This approach is not new and can be traced back to the work of Clark in 1979. However it seems that it has been - to a certain extent - forgotten.

The paper deals with partial correctness, we complement it with an outline of a method for proving completeness.

In this paper we recall a simple and straightforward method of proving partial correctness properties of definite clause logic programs. The method is declarative: it treats programs as sets of axioms and the computed instances of queries as logical formulae; it abstracts from any operational semantics. The method may seem well known, it is discussed among others in [Cla79, Hog81, Hog84, Der93]. However some publications on the theory of logic programming, for instance [Apt97, PR97], use an unnecessarily complicated approach for proving such properties. That approach is operational, it refers to LD-resolution and considers the procedure calls that occur during the computation.

A main advantage of logic programming is that for many purposes one can abstract from the "control" of programs and consider only their "logic". Using an operational approach to prove program properties (such that can be dealt with in a declarative way) weakens this advantage. In this paper we show that, as long as we are not interested in the form of procedure calls during the computation, the declarative proof method is simpler and not less general than the operational one. We show that in a certain sense it is more powerful.

First we present the declarative proof method (which will be called "natural"), then the operational one. In the third section we compare the two approaches. In particular we show that whatever can be proved by the operational method, can also be proved by the declarative one (as long as properties of program answers

[1]Institute of Computer Science, Polish Academy of Sciences and IDA, Linköpings universitet, Sweden. E-mail: `drabent@ipipan.waw.pl`

are considered). The next section relates the natural proof method to so called annotation method.

The subject of this paper is a declarative approach to proving partial correctness of logic programs. In the appendix we briefly present a corresponding approach to proving completeness.

1 Preliminaries

For basic definitions etc. we refer the reader to the standard references and to [Apt97]. We use a variant of SLD-resolution with queries (of the form $A_1, \ldots, A_n$) instead of goals (of the form $\leftarrow A_1, \ldots, A_n$). By a computed (resp. correct) answer we mean an instance $Q\theta$ of a query Q, where θ is a computed (correct) answer substitution for Q and the given program.

We are interested in declarative properties of programs, i.e. properties of programs treated as sets of logic formulae. Speaking more formally, we consider properties of programs' (computed or correct[2]) answers. So we do not distinguish programs which are logically equivalent but have different S-semantics.

2 Declarative specifications and the natural proof method

As a standard example let us take the program APPEND:

```
app([],L,L)
app([H|K],L,[H|M]) :- app(K,L,M)
```

We want to prove that it indeed appends lists. We have to begin with a precise statement (a specification) of this property. A slight complication is that the program does not actually define the relation of list concatenation, but its superset. This is because its least Herbrand model contains "ill-typed" atoms, like $app([\,],1,1)$.

So we want to prove that:

For any answer $app(k,l,m)$, if k and l are lists then m is a list and $k * l = m$. (1)

(By a list we mean a term $[t_1, \ldots, t_n]$ (in Prolog notation), where $n \geq 0$ and $t_1, \ldots, t_n$ are possibly non-ground terms. Symbol $*$ denotes the concatenation of lists[3].)

[2] By the soundness and completeness of SLD-resolution, for a given program the sets of computed and of correct answers are equal.

[3] Actually, the requirement on k is unnecessary. Our intention however is to follow the corresponding example from [Apt97]. A full specification of APPEND may be: if l is a list or m is a list then k, l, m are lists and $k * l = m$.

This specification could be equivalently expressed as

$$spec \models app(k,l,m) \tag{2}$$

for any answer $app(k,l,m)$, where $spec$ is the Herbrand interpretation:

$$spec = \{\, app(k,l,m) \in \mathcal{H} \mid \text{if } k \text{ and } l \text{ are lists then } m \text{ is a list and } k * l = m \,\}$$

($\mathcal{H}$ is the Herbrand base; we assume a fixed infinite set of function symbols). Obviously, (2) holds iff all the ground instances of $app(k,l,m)$ are in $spec$.

Notice that we do not need to refer to the notion of a query in the specification. Assume that $app(k,l,m)$ is a success instance of query $app(k',l',m')$. If k',l' are lists then (k,l are lists and) the specification implies that m is a list and $k*l=m$.

Such specifications, referring to program answers, will be called *declarative*. A declarative specification can be an interpretation (possibly a non Herbrand one) or a set of axioms[4]. In this paper we will use specifications of the first kind. We will say that a program is *correct* w.r.t. a declarative specification $spec$ iff $spec \models Q$ for any answer Q of the program.

To prove the partial correctness (of a logic program w.r.t. a declarative specification) we use an obvious approach, discussed among others by Clark [Cla79], Hogger [Hog81, p. 378–9] and Deransart [Der93, Section 3][5]. We will call it the *natural* proof method:

> Let P be a program and $spec$ be an interpretation. To show that $spec \models Q$ for every computed/correct instance Q of a query it is sufficient to show that $spec \models P$.

So we have to show that $spec \models C$ for each clause C of the program. This proof method is obviously sound (as $P \models Q$, by soundness of SLD-resolution). It is also complete [Der93] in the following sense. If a program P is correct w.r.t. a declarative specification $spec$ then there exists a stronger specification $spec' \subseteq spec$ such that $spec' \models P$ (and thus the method is applicable to $spec'$).

In our example the proof is simple. We present here the less trivial part with details. Consider the second clause. To show that

$$spec \models app([H|K],L,[H|M]) \leftarrow app(K,L,M)$$

take ground terms h,k,l,m[6] such that $spec \models app(k,l,m)$. (In other words $app(k,l,m) \in spec$). Assume that $[h|k]$ and l are lists (hence k is a list). Then m is

[4] An axiomatic specification equivalent to our example specification may consist of formula $app(k,l,m) \leftrightarrow (list(k), list(l) \rightarrow list(m), k{*}l{=}m)$ together with axioms describing predicates $=$ and $list$ and function $*$.

[5] where it is called "inductive proof method".

[6] and valuation $\{\, H/h, K/k, L/l, M/m \,\}$

a list and $k * l = m$, as $spec \models app(k, l, m)$. Thus $[h|m]$ is a list and $[h|k] * l = [h|m]$. We showed that $spec \models app([h|k], l, [h|m])$, this concludes the proof.

We present another simple example. Consider the standard REVERSE program which uses the so called accumulator technique.

```
reverse( X, Y ) :- rev( X, Y, [] ).
rev( [], X, X ).
rev( [H|L], X, Y ) :- rev( L, X, [H|Y] ).
```

The declarative reading of the program is simple: the first argument of *rev* is a list, its reverse is represented as a difference list, of the second and the third argument. The formal specification is

$$\begin{array}{rcl} spec & = & \{reverse([t_1, \ldots, t_n], [t_n, \ldots, t_1]) \mid n \geq 0, t_1, \ldots, t_n \in \mathcal{T}\} \\ & \cup & \{rev([t_1, \ldots, t_n], [t_n, \ldots, t_1|t], t) \mid n \geq 0, t_1, \ldots, t_n, t \in \mathcal{T}\} \end{array}$$

where $\mathcal{T}$ is the set of ground terms. The nontrivial part of the proof is to show that the last clause is true in the interpretation *spec*. Take ground terms l, x, h, y, such that $spec \models rev(l, x, [h|y])$. So there exist $n \geq 0, t_1, \ldots, t_n, t$ such that $l = [t_1, \ldots, t_n]$, $x = [t_n, \ldots, t_1|t]$, $t = [h|y]$. Then $rev([h|l], x, y)$ is $rev([h, t_1, \ldots, t_n], [t_n, \ldots, t_1, h|y], y)$, thus $spec \models rev([h|l], x, y)$.

Notice that the natural method refers only to the declarative semantics of programs. A specification is an interpretation. Correctness is expressed as truth (of the program's answers) in the interpretation. Program clauses are treated as logic formulae, their truth in the interpretation is to be shown[7]. We abstract from any operational semantics, in particular from the notions of a selected literal and the selection rule used by resolution. Still we can use declarative specifications to reason about queries and corresponding answers, using the fact that an answer is an instance of the query.

3 Call-success specifications and the operational approach

Some authors [BC89], [Apt97, Chapter 8], [PR97][8] propose another approach to proving partial correctness of logic programs. The main difference is that they specify the properties of interest in another, non-declarative way. To deal with "ill-typed" atoms they consider the form of queries. They use specifications consisting of two parts. The *precondition* specifies the "procedure calls" that appear during the computation (more precisely, the atoms that are selected in the LD-resolution). The

[7]Alternatively we may consider an axiomatic specification of the intended property (by means of a set of axioms T) and show that the program is a logical consequence of T.

[8]Whenever these approaches differ, we follow that of [Apt97].

postcondition specifies the procedure successes (the computed instances of procedure calls). We will call such specifications *call-success specifications*. Formally, pre- and postconditions are sets of atoms, closed under substitutions[9]. Hence, if a procedure call satisfies the precondition then also any corresponding success does.

A program is correct if every procedure call and every success satisfy the pre- or postcondition respectively[10]. Notice that this is not a declarative property. It considers computations (SLD-trees), not only computed answers, and it depends on the selection rule used. Prolog selection rule is assumed. The proof method used was proposed by Bossi and Cocco [BC89] and is an instance of the method of Drabent and Małuszyński [DM88]. We will call it the *operational* proof method.

The method is based on the following verification condition. For each clause C of the program, show that for each (possibly non-ground) instance $H \leftarrow B_1, \ldots, B_n$ ($n \geq 0$) of C

if $H \in pre,\ B_1, \ldots, B_k \in post$ then $B_{k+1} \in pre$ (for $k = 0, \ldots, n-1$),
if $H \in pre,\ B_1, \ldots, B_n \in post$ then $H \in post$.

The condition on the initial query is that, for every its instance $B_1, \ldots, B_n$ ($n > 0$), if $B_1, \ldots, B_k \in post$ then $B_{k+1} \in pre$ (for $k = 0, \ldots, n-1$). In the terminology of [Apt97], a program satisfying the verification condition is called, together with its specification, well-asserted.

So the operational method requires proving one implication per an atom occurring in the program or in the initial goal. In contrast, the natural method advocated in this paper requires proving one implication per program clause.

Let us come back to our example. We refer here to its treatment in [Apt97, p. 214]. The precondition is

$$pre = \{\, app(k, l, m) \mid k \text{ and } l \text{ are lists} \,\}.$$

(Here k, l, m are terms, possibly non-ground. An atom A satisfies pre if $A \in pre$). The postcondition is

$$post = \{\, app(k, l, m) \mid k, l, m \text{ are lists and } k * l = m \,\}.$$

The verification conditions to be proved consist of one implication for the first clause of APPEND and two implications for the second one. The details of the proof can be found in [Apt97].

The reader can see that that proof is more complicated than the one using the natural method. This should not be surprising as the proved property is stronger. In addition to the required property, we also get a proof that for a certain class of initial goals each atom selected by LD-resolution satisfies the precondition. However, the latter property is often of no interest.

[9]If an atom satisfies the precondition then so do all its instances.
[10]Provided that the initial goal satisfies a certain condition.

We should also mention that the method is complete: Assume that a program P with a set QQ of initial queries is correct w.r.t. a call-success specification $\langle pre, post \rangle$. Then there exists a stronger specification with $pre' \subseteq pre$ and $post' \subseteq post$ such that P and $\langle pre', post' \rangle$ satisfy the verification condition and the queries in QQ satisfy the condition on initial queries. The main idea of a proof of this fact [Dra88] is to consider the LD-derivations starting from queries from QQ and to take as pre' and $post'$ the sets of all the procedure calls and successes, respectively, occurring in these derivations.

4 Comparison

In this section we first compare the specification styles of the two approaches. Then we show that, although formally both methods are equivalent, the declarative one is, in a certain sense, stronger.

A declarative specification refers to *all* the answers of the program (i.e. success instances of arbitrary queries). It is independent from the operational semantics. A call-success specification refers to a particular operational semantics (LD-resolution) and to computations starting from a *restricted* class of initial queries. The specification consists of a precondition and a postcondition. The precondition refers to the selected atoms in LD-derivations; the postcondition to their success instances.

A call-success specification is a pair of sets of atoms; a declarative specification is an interpretation over an arbitrary universe, or it is an axiomatic theory.

A common phenomenon in logic programming is that a program's semantics contains atoms which are irrelevant for the correctness of the program. For instance, APPEND program is intended to define the relation of appending lists. But it actually defines a superset of this relation; its least Herbrand model contains atoms $app(k, l, m)$ where l and m are not lists[11].

The two compared approaches deal with such "ill-typed" atoms in different ways. In the natural method, the specification is a superset of the least Herbrand model M_P. Such specifications could be seen as (conjunctions of) implications; we just do not bother and *include* all "ill-typed" atoms. In the other approach, "ill-typed" atoms are *excluded* from the specification. The precondition pre is used for this purpose. The postcondition is not (in general) a superset of M_P. Instead it is a superset of $M_P \cap pre$. We may say that a declarative specification specifies M_P while the postcondition of a call-success specification specifies $M_P \cap pre$.

Now we are going to show that, given an call-success specification, one can construct a declarative specification which is, in a sense, equivalent. Consider an operational specification $\langle pre, post \rangle$. A corresponding declarative specification used in our approach could be seen, speaking informally, as implication $pre \rightarrow post$.

[11] A technical detail: We assume that the Herbrand universe contains a term which is not a list.

Definition 4.1 Let *pre* and *post* be sets of atoms closed under substitution. The *declarative specification corresponding* to the call-success specification $\langle pre, post\rangle$ is the Herbrand interpretation

$$pre{\rightarrow}post \ := \ \{\, A \in \mathcal{H} \mid \text{if } A \in pre \text{ then } A \in post \,\}$$

In other words, $pre{\rightarrow}post = (\mathcal{H} \setminus pre) \cup (\mathcal{H} \cap post)$. We have:

Proposition 4.2 If a program P is correct w.r.t. the call-success specification $\langle pre, post\rangle$ then P is correct w.r.t. declarative specification $pre{\rightarrow}post$.

If P is correct w.r.t. $pre{\rightarrow}post$ and $A\theta$ is an answer to a query $A \in pre$ then $A\theta \in post$.

PROOF: The second part is obvious. For the first part assume that a program P is correct w.r.t. $\langle pre, post\rangle$. So for any atomic query from *pre* any its computed instance is in *post*. Consider any computed instance A of an arbitrary atomic query and assume that $pre{\rightarrow}post \not\models A$. Then some ground instance A' of A is not a member of $pre{\rightarrow}post$. This means that $A' \in pre$ and $A' \notin post$. As A is a computed instance of a query, query A succeeds with an empty answer substitution. Any instance of A has the same property; thus A' succeeds and from the assumption we obtain $A' \in post$, contradiction. This completes the proof. □

Proposition 4.3 Assume that it can be shown, by the method of [BC89], that a program P is correct w.r.t. a call-success specification $\langle pre, post\rangle$. Then it can be shown that P is correct w.r.t. declarative specification $pre{\rightarrow}post$, using the natural method.

In other words, if P and $\langle pre, post\rangle$ satisfy the verification condition of the operational method then $pre{\rightarrow}post \models P$.

PROOF (see also [CD88]): As $pre{\rightarrow}post$ is a Herbrand interpretation, $pre{\rightarrow}post \models P$ is equivalent to a property that if $B_1, \ldots, B_n \in pre{\rightarrow}post$ then $H \in pre{\rightarrow}post$, for any ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause of P,

To show the latter, assume that $B_1, \ldots, B_n \in pre{\rightarrow}post$. If $H \notin pre$ then $H \in pre{\rightarrow}post$. Otherwise, if $H \in pre$ then by simple induction we obtain from the verification condition that $B_i \in pre$ and $B_i \in post$, for $i = 1, \ldots, n$. Hence, by the verification condition, $H \in post$, thus $H \in pre{\rightarrow}post$. □

The two propositions show that the natural method is stronger than the operational method, as far as declarative properties are concerned. Moreover, a declarative specification corresponding to a given call-success specification is obtained from the latter by a simple composition of three operations: removing non-ground atoms, set complementation and set union.

Formally, the natural method is not strictly stronger, as it may be treated as a special case of the operational method. (For a given declarative specification *spec*, take the set of all atoms as the precondition and $post = \{A \mid spec \models A\}$ as the postcondition). However we want to view the operational method together with its style of specifications which uses nontrivial preconditions and postconditions which are not supersets of M_P and which do not contain "ill-typed" atoms. The operational method understood in this way is in a sense less powerful than the declarative method.

First, correctness w.r.t. the declarative specification $pre{\rightarrow}post$ does not imply correctness w.r.t. the call-success specification $\langle pre, post\rangle$. The latter requires that all the atoms selected in LD-resolution are in pre. So the call-success specification above is strictly stronger than the declarative specification. Often a desired property is conveniently expressed by a declarative specification $pre{\rightarrow}post$ (or by intersection of some such specifications) but the selected atoms are not in pre. In such a case the operational method is inapplicable (conf. [BM97]). We illustrate this by an example, where under any selection rule the precondition does not hold

Let us consider the following program P.

```
p(X,Y) :- q(X,W,Z,Y),q(Z,_,W,_).
q(X,Y,X,Y).
```

This program is artificial, however it is an abstraction of "two-pass" programs and of certain usages of difference lists. (Some examples of such programs could be found e.g. in [BM97]). Let $\lambda(t)$ stands for "t is a list", for a possibly non-ground term t. Let

$$
\begin{array}{ll}
pre_p = \{\, p(t,s) \mid \lambda(t)\,\} & post_p = \{\, p(t,s) \mid \lambda(s)\,\} \\
pre_q = \{\, q(t,s,u,v) \mid \lambda(t)\,\} & post_q = \{\, q(t,s,u,v) \mid \lambda(u)\,\} \\
pre'_q = \{\, q(t,s,u,v) \mid \lambda(s)\,\} & post'_q = \{\, q(t,s,u,v) \mid \lambda(v)\,\} \\
pre = pre_p \cup (pre_q \cap pre'_q) & post = post_p \cup (post_q \cap post'_q).
\end{array}
$$

The program is correct w.r.t. declarative specification $pre{\rightarrow}post$. It is also correct w.r.t. a stronger declarative specification

$$spec = (pre_p{\rightarrow}post_p) \cap (pre_q{\rightarrow}post_q) \cap (pre'_q{\rightarrow}post'_q).$$

Moreover, $spec \models P$ and the natural method is applicable[12]. On the other hand,

[12]We present here in details the nontrivial part of the proof, showing that the first clause of P is true in $spec$. Take a ground instance $H \leftarrow B_1, B_2$ of the clause. Notice that:

$$
\begin{array}{lcl}
H \in pre_p & \text{implies} & B_1 \in pre_q \\
B_1 \in post_q & \text{implies} & B_2 \in pre_q \\
B_2 \in post_q & \text{implies} & B_1 \in pre'_q \\
B_1 \in post'_q & \text{implies} & H \in post_p .
\end{array}
\tag{3}
$$

Assume that the right hand side is true in $spec$: $spec \models B_1, B_2$. Thus we have:

$$
\begin{array}{lcll}
B_i \in pre_q & \text{implies} & B_i \in post_q & \text{for } i = 1,2 \\
B_1 \in pre'_q & \text{implies} & B_1 \in post'_q &
\end{array}
$$

Combining these implications together we obtain that from our assumption it follows that $H \in pre_p$

the program is obviously not correct w.r.t. call-success specification $\langle pre, post\rangle$ and the operational method is inapplicable[13].

In contrast to the case of finding a declarative specification corresponding to a given call-success one, the reverse mapping does not exist. For a given declarative specification, there does not exist a call-success specification (with a non-trivial precondition) such that if P is correct w.r.t. the former then P is correct w.r.t. the latter. Such a call-success specification depends on (the behaviour under LD-resolution of) P.

So although both methods are equivalent (if declarative properties are concerned), the natural method is strictly stronger in the following sense. There exists a (rather simple) mapping transforming a call-success specification into a declarative one, equivalent in the sense of Propositions 4.2 and 4.3. Such a mapping from declarative specifications into call-success ones (with nontrivial preconditions) does not exist.

Informal comparison suggests that the declarative specifications are simpler and more natural. The verification conditions of the natural method consist of substantially fewer implications to prove (conf. the previous section). So, at least in most cases, the proofs using the natural method are simpler.

5 A note on related work

In this section we briefly compare the natural method with with the annotation method of Deransart [Der93, Section 4], [BM97, Section 4] of proving declarative properties of logic programs. Then we mention some other related work.

A specification in the natural method is usually in a form of an implication (or a conjunction of implications). The annotation method treats the specification in a different way. It treats the antecedents and consequents of the implications in the specification as separate properties. Let us call them elementary properties. One assigns directions ("inherited" or "synthesised") to these properties. Then methods of attribute grammars are used to construct a sufficient condition for program correctness.

Applying this approach to our last example, the specification consists of inherited properties pre_p, pre_q, pre'_q and synthesised properties $post_p, post_q, post'_q$. As a sufficient condition to prove, one obtains the implications (3).

The annotation method could be seen as a refinement of the natural method, with a lower granularity. The verification condition of the natural method is a set

implies $H \in post_p$. This means that $spec \models H$. Thus we showed that for an arbitrary ground instance C of the clause, $spec \models C$, which completes the proof.

[13] P is correct w.r.t. call-success specification $\langle (pre_p \cup pre_q), (post_p \cup post_q)\rangle$ but this specification is too weak to construct a proof by the operational method. This can be done if the postcondition expresses the fact that the second and the fourth arguments of q are equal, or that if the second argument is a list then the fourth is a list. The latter idea results in the specification $\langle\, pre_p \cup pre_q\ ,\ post_p \cup (post_q \cap \{A \mid pre'_q \rightarrow post'_q \models A\})\,\rangle$.

of "big" implications. The approach of [Der93, BM97] shows how to decompose them into smaller ones. That approach could also be understood as studying the structure of the proofs in the natural method.

It could be seen as a disadvantage of the annotation method that its style of specification hides the information about which antecedent implies which consequent. (For instance, such specification for our example does not state that $post_q$ depends on pre_q but not on pre'_q). Also that approach involves a bigger technical apparatus. (It requires establishing an ordering among the elementary properties of the atoms of each clause. It also requires showing non-circularity of some attribute grammar.)

We should mention the work of Stärk [Stä97] on proving properties of Prolog programs. It considers programs with negation and contains an implementation of a theorem prover/checker. It deals with a broader class of properties than our work, for instance program termination. The purpose of this paper is different; we want to recall a simple and basic method and show that it is at least as useful as the operational one. Also Naish [Nai96] advocates using the declarative semantics, instead of operational, in reasoning about logic programs.

6 Conclusions

We presented a simple and natural way of proving declarative partial correctness properties of definite clause programs. The method is not new and can be traced back at least to [Cla79]. It is based on a property that if $SPEC \models P$ then, for any computed by program P instance Q of a query, $SPEC \models Q$. We discussed the case when the specification $SPEC$ is an interpretation. However it can also be a set of axioms. We compared the natural method with the approach used in [Apt97, PR97], which is based on operational semantics (LD-resolution). We showed that whatever can be proved by the operational method, can be proved by the natural one. We showed that in a certain sense the natural method is strictly stronger than the operational one.

The author believes that the declarative method (possibly treated informally) is a valuable tool for programmers in their every-day reasoning about programs, and that it is actually used by many of them. It should be included in teaching logic programming.

If it were necessary to resort to operational semantics in order to prove simple program properties then logic programming would not deserve to be called a declarative programming paradigm. This work shows that this is not the case.

ACKNOWLEDGEMENTS

Thanks are due to Pierre Deransart for discussions on the subject. Krzysztof Apt suggested writing this report.

References

[Apt90] K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 10, pages 493–574. Elsevier Science Publishers B.V., 1990.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89, vol. 2*, pages 96–110. Springer-Verlag, 1989. Lecture Notes in Computer Science.

[BM97] J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, 1997.

[CD88] B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming. *Information and Computation*, 78(1):1–55, 1988.

[Cla79] K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DM88] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59:133–155, 1988.

[Dra88] W. Drabent. On completeness of the inductive assertion method for logic programs. Unpublished note, Institute of Computer Science, Polish Academy of Sciences, May 1988.

[Hog81] C. J. Hogger. Derivation of logic programs. *Journal of ACM*, 28(2):372–392, 1981.

[Hog84] C. J. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

[Nai96] L. Naish. A declarative view of modes. In *Proceedings of JICSLP '96*, pages 185–199. MIT Press, 1996.

[PR97] D. Pedreschi and S. Ruggieri. Verification of logic programs. Technical Report TR-97-05, Department of Computer Science, University of Pisa, 1997.

[Stä97] R. F. Stärk. Formal verification of logic programs: Foundations and implementation. In *Logical Foundations of Computer Science LFCS '97 — Logic at Yaroslavl*, pages 354–368. Springer-Verlag, 1997. LNCS 1234.

Appendix. A note on proving completeness

The main subject of this paper is the natural method of proving partial correctness of logic programs. Here we briefly discuss a method for proving completeness.

For a given program, a specification for completeness is in general different from that for partial correctness. Specifications considered in the former sections describe supersets of programs' sets of answers. A specification for completeness describes a subset. For such a specification *cspec* we want to prove that if $cspec \models Q$ then Q is an answer for the program. As previously, we consider specifications which are (possibly non Herbrand) interpretations. (Alternatively, we may consider specifications which are theories). We require that these interpretations are models of Clark equality theory CET.

For example, a specification for completeness of APPEND may be the Herbrand interpretation

$$cspec = \{ app(k,l,m) \in \mathcal{H} \mid k,l,m \text{ are lists, } k*l = m\} \cup \{t{=}t \mid t \text{ is a ground term}\}.$$

Let P be a definite clause program. Below we refer to theory ONLY-IF(P) that is usually used when defining Clark completion comp(P) of a program. Informally, ONLY-IF(P) is P with implications reversed. For the definition the reader is referred e.g. to [Apt90]. In our example, ONLY-IF(APPEND) is

$$app(x,y,z) \;\rightarrow\; x = [\,], y = z \;\vee\; \exists h,k,l,m\colon x = [h|k], y = l, z = [h|m], app(k,l,m)$$

The following property can be used to prove completeness of a program.

Proposition 6.1 Let P be a program and Q a query. Assume that

(i) $cspec \models$ ONLY-IF(P) and $cspec \models$ CET,

(ii) P terminates for Q, i.e. there exists a finite SLD-tree for Q and P.

Then

1. if $cspec \models \exists Q$ then $P \models \exists Q$ (and some instance of Q is an answer of P),
2. if $cspec \models Q$ then $P \models Q$ (and Q is an answer of P).

PROOF (outline)

1. By induction on the SLD-tree. Let $Q_1,\ldots,Q_n$ ($n \geq 0$) be the children of the root Q in the SLD-tree. As $cspec \models \exists Q$, by (i) we have $n > 0$ and $cspec \models \exists Q_i$ for

some i. Now either Q_i is empty and then Q succeeds, thus $P \models \exists Q$ or Q_i is not empty and $P \models \exists Q$ by the inductive assumption.

2. If *cspec* $\models Q$ then, by 1., $P \models Q'$ for any ground instance Q' of Q. Hence $P \models Q$ (under an assumption that P is finite and the set of function symbols in the language is infinite).

□

EXAMPLE Consider program APPEND and the specification *cspec* given above. It is easy to show that *cspec* $\models$ ONLY-IF(APPEND). Consider $Q = app(k, l, m)$ where m is a list. One can show, using any standard method, that Q terminates under Prolog selection rule.

Now assume that k, l are variables. Then *cspec* $\models \exists Q$ and we obtain from the proposition that $P \models \exists Q$. Taking k', l' being lists such that $k' * l' = m$ we have *cspec* $\models app(k', l', m)$ and from the proposition we get $P \models app(k', l', m)$. So, by completeness of SLD-resolution, Q succeeds and produces all the required divisions of m into two lists.