Verifying correctness of logic programs

A. Bossi, N. Cocco
Dip. di Matematica Pura ed Applicata - Univ. di Padova - Italy.

Abstract

We are convinced that logic programming needs specifications as much as traditional programming. For this reason, it is necessary to study also for logic programs how correctness with respect to a given specification can be asserted. Starting from Hogger's considerations on this topic, we supply a sufficient condition for completeness. Modularity and pre/post specifications of program modules are then discussed. We propose a sufficient condition for partial correctness with respect to a pre/post specification. Some small examples illustrate our technique.

1. Introduction

One reason of the superiority of logic programming over traditional programming, would be that the program coincides with its specification. This claim is based on the fact that the Horn-clause notation, which is a subset of first-order predicate calculus, is used as a programming language. One simple objection to this assertion is that extra-logical features (like cut, fail, etc.) are normally used in order to obtain an efficient implementation. But even considering logic programming, independent from the implementation and without extra-logical features, we believe that the claim is exaggerated. In fact definite Horn clauses are a rather limited notation. Their major weakness consists in the lack of negation, which leads to very complicated descriptions. A common solution to overcome the absence of the not operator is to explicitly define another predicate for the negated one, as in the following simple example:

1) dominate([], []).
   dominate([a|s], [b|v]) :- (a≥b), dominate(s, v).
2) nondom([a|s], []).
   nondom([], [b|v]).
   nondom([a|s], [b|v]) :- (a<b).
   nondom([a|s], [b|v]) :- (a≥b), nondom(s, v).

where dominate(x, y) clearly means that the two lists x and y have the same length and that all the elements in x are greater or equal to the corresponding ones in y. Less clear is the fact that nondom(x, y) ≡ ¬ dominate(x, y) !

Moreover, since efficiency has to be taken into consideration, often simplicity is sacrificed to it. An example is the very common use of the accumulation technique. Consider the following simple program:

reverse (X, Y) :- rev(X, [], Y).
rev([], Y,Y).
rev([A|X], Y, Z) :- rev(X, [A|Y], Z).

where the reverse of a list is computed by using the accumulation technique. The implementation choice is embodied into the declarative semantics of the program, thus making it more obscure.

All this leads to complicated and unnatural predicate definitions. In order to understand the meaning of a predicate, it is often necessary to read many Horn clauses and to understand the meaning of many other predicates used in the definition. The relation between such predicates and clauses is often not a natural one. For these reasons often the declarative semantics of a program is not a satisfactory specification . This is particularly true for larger programs which require a modular design; then clean interfaces are needed to hide particular implementation choices. As it happened in procedural programming, more abstract specifications would help in a modular approach to programming, top-down design, proving general properties and software reusability.

A specification is often used to express the intended meaning of the program, so that the correctness with respect to it, just means correctness with respect to such an intended meaning. Clearly this requires that the specification language allows for a natural formalization of the intended meaning. With regard to this, we believe that a very important requirement is the ease in expressing data and their manipulations. Although a logic program actually handles terms of the Herbrand basis, these correspond to objects in structured domains, such as natural numbers or lists. Hence we think it is much clearer to use their standard operators and properties in the specification.

In the field of imperative languages, the importance of a formal and precise description of what a program is to do and the need for a methodology in order to insure that it actually behaves, have been discussed and studied since long time and they are presently widely recognized. Up to now, in the logic programming field, the use of specifications has been proposed mainly for program synthesis. The analogies with functional languages are evident. On the functional side [Bur77], programs are synthesized starting from the descriptions of mathematical functions. On the logic side [Hog81, Sato84], the specification, in a first order language, is transformed into an executable logic program. This produces an initial program, while efficiency requirements force further optimizations. These can be achieved by transformation techniques which preserve correctness while improving efficiency, see for example [Bos87, Fut87, Tam 84, Gal 86]. The development of practical and theoretically well-founded methodologies to help the programmer in synthesizing correct and efficient programs from their specifications, is undoubtedly to be wished. Nevertheless, methods and tools for program verification are still necessary, since a more practical approach in program design is, in our opinion, a skilful blend of synthesizing and assembling already produced and verified modules.

In this paper we intend to examine the problem of verifying the correctness of a logic program with respect to a given specification. This topic has been studied by Clark and Tarnlund [Cla77], Balogh [Bal78], Hogger [Hog81, Hog84] and, more recently, by Drabent and Maluszynski [Dra87]. In [Cla77] a first order logic theory is developed for axiomatizing data structures and computable relations on them. Logic programs are then derived from such relations and their partial correctness and termination can be stated in the theory. [Bal78] defines partial correctness of a logic program with respect to pre- and post-assertions. Post-assertions are of two kinds: either necessary or sufficient to the fulfillment of the relation corresponding to the program. Rules for partial correctness are also defined for Prolog programs. Hogger

[Hog81, Hog84] distinguishes between partial correctness and completeness of a logic program with respect to a given specification. He gives also a sufficient criterion for partial correctness. His work developed from the area of program synthesis. In [Dra87] a method is presented which is directly inspired by the results of Floyd [Flo67] and Hoare [Hoa69] for imperative programming. Each program predicate is annotated with a couple of assertions, the pre- and the post-condition, which specify properties which the program has to satisfy. The authors do not consider general specifications, they are rather interested in proving run-time properties, such as binding of parameters or modes.

In the following we consider only logic programs, that is sets of definite Horn clauses with complete resolution. In section 2 we introduce program specifications as extensions of the theory of data domains. Starting from the definitions given by Hogger in [Hog81, Hog84], we show how total correctness of a logic program with respect to a given specification can be proved by means of sufficient conditions. The sufficient condition for partial correctness was given by Hogger, while we supply another one for completeness. In section 3 we consider also weaker specifications, like general properties of program modules, which are the ones usually interesting for modular design and program transformation. They can be expressed in terms of pre/post conditions of program modules. We supply a sufficient condition for partial correctness with respect to a pre/post specification. All this is illustrated by simple examples.

2. Correctness of logic programs

We consider purely logic programs, which means that a program is defined to be just a set of definite Horn clauses

$$\forall\, y_1, \ldots, y_h.(A \leftarrow A_1 \wedge \ldots \wedge A_m) \ , \ m \geq 0 \ ,$$

where A, A_i , with $1 \leq i \leq m$, are atomic formulas of the form $p(t_1,\ldots,t_k)$, $k>0$, consisting of a k-ary predicate symbol p and k terms $t_1,\ldots,t_k$ built up in the standard way. The terminology adopted will be the standard one for logic programming.

A program specification is generally used for describing the intended meaning of a program, that is what the programmer intended to define, without caring of how it has been realized. Both data and operations on them, should be described as abstractly as possible, while pointing out their typical properties. In particular, it is not useful to represent the data handled by the program as terms of the Herbrand universe. In the programmer's mind they have a type, a structure and they obey laws which are evident to him. Actually it is often in virtue of these properties of data that the programmer convinces himself of a program correctness. Therefore in our opinion, a specification, in order to be a self convincing description of the intended meaning of the program, should be based on the intended data domain.

Let $\mathcal{P}$ be a program, $\mathcal{S}_0$ the theory which describes the data domains and $L_{\mathcal{S}0}$ the associated language. Then the language $L_{\mathcal{P}}$ associated to the program $\mathcal{P}$ is $L_{\mathcal{S}0}$ plus the predicate symbols in $\mathcal{P}$. Let us assume that for each predicate symbol p in $\mathcal{P}$ there is a corresponding new predicate symbol $p_{\mathcal{S}}$ defined in the specification. A specification language $L_{\mathcal{S}}$ for $\mathcal{P}$ is $L_{\mathcal{S}0}$ augmented with such predicate symbols corresponding to the ones appearing in $\mathcal{P}$. A specification for $\mathcal{P}$ consists of a set of definitions

$$\forall\, x_1, \ldots, x_k. \ (p_{\mathcal{S}}(x_1, \ldots, x_k) \ \leftrightarrow \ \mathcal{S}_p(x_1, \ldots, x_k)) \qquad \text{(i)}$$

where p is a k-ary predicate symbol appearing in $\mathcal{P}$ and $\mathcal{S}_p(x_1, \ldots, x_k)$ is a first order formula of $L_{\mathcal{S}}$.
We denote by $\mathcal{S}$ the first order theory which extends the domain theory $\mathcal{S}_0$ with the axioms (i), which define the intended meaning of $\mathcal{P}$.

The meaning of p in $\mathcal{S}$ is defined as the specified relation associated to $p_{\mathcal{S}}$ in $\mathcal{S}$:

$\mathcal{M}(p)_{\mathcal{S}} =_{def} \{(t_1, \ldots, t_k) \mid t_i$ ground terms and $\mathcal{S} \vdash p_{\mathcal{S}}(t_1, \ldots, t_k)\}$

where $\mathcal{S} \vdash$ means provability in first-order predicate logic (for example by natural deduction).

On the other hand for each k-ary predicate symbol, p, the meaning of p in $\mathcal{P}$ is defined as the computed relation associated to p in $\mathcal{P}$:

$\mathcal{M}(p)_{\mathcal{P}} =_{def} \{(t_1, \ldots, t_k) \mid t_i$ ground terms and $\mathcal{P} \vdash p(t_1, \ldots, t_k)\}$.

Following the definitions given by Hogger in [Hog81, Hog84], we distinguish partial correctness, completeness and total correctness of $\mathcal{P}$ with respect to $\mathcal{S}$.

$\mathcal{P}$ is partially correct with respect to $\mathcal{S}$ iff the relations defined by $\mathcal{P}$ (computed relations) are included in those defined by $\mathcal{S}$ (specified relations):

$\forall p. \mathcal{M}(p)_{\mathcal{S}} \supseteq \mathcal{M}(p)_{\mathcal{P}}$.

$\mathcal{P}$ is complete with respect to $\mathcal{S}$ iff the relations defined by $\mathcal{S}$ (specified relations) are included in those defined by $\mathcal{P}$ (computed relations):

$\forall p. \mathcal{M}(p)_{\mathcal{P}} \supseteq \mathcal{M}(p)_{\mathcal{S}}$.

$\mathcal{P}$ is totally correct with respect to $\mathcal{S}$ iff it is both partially correct and complete with respect to $\mathcal{S}$. This means that the relations defined by $\mathcal{P}$ are exactly the same relations defined by $\mathcal{S}$:

$\forall p. \mathcal{M}(p)_{\mathcal{P}} = \mathcal{M}(p)_{\mathcal{S}}$.

2.1. Partial correctness

The proof of partial correctness of a program $\mathcal{P}$ with respect to a specification $\mathcal{S}$ can be given, clause by clause, using the following sufficient condition:

$\mathcal{P}$ is partially correct with respect to $\mathcal{S}$ if for any clause c in $\mathcal{P}$: $\mathcal{S} \vdash c_{\mathcal{S}}$ (1)

where $c_{\mathcal{S}}$ indicates the clause c in which the predicates in $\mathcal{P}$ are replaced by the corresponding ones in $\mathcal{S}$. This corresponds to proving, for each definite Horn clause c in $\mathcal{P}$, that $c_{\mathcal{S}}$ is a theorem in the theory defined by $\mathcal{S}$, that is the theory containing the definitions of the specified predicates, based on the theories for the data types. Thus, since we assumed that each predicate symbol in $\mathcal{P}$ has a corresponding specification in $\mathcal{S}$, proving $\mathcal{S} \vdash c_{\mathcal{S}}$, with

c: $\forall y_1, \ldots, y_h. (A \leftarrow A_1 \wedge \ldots \wedge A_m)$, $0 \leq m$,

actually means to prove:

$\mathcal{S} \vdash (A_{\mathcal{S}} \leftarrow A_{1\mathcal{S}} \wedge \ldots \wedge A_{m\mathcal{S}})$, $0 \leq m$,

where if $P = p(x_1, \ldots, x_k)\tau$, with τ substituting terms for variables, then $P_{\mathcal{S}} =_{def} p_{\mathcal{S}}(x_1, \ldots, x_k)\tau$.

This sufficient criterion was given by Hogger in [Hog81, Hog84], we just stressed the fact that each clause can be proved separately. This obviously simplifies the proof of partial correctness.

Example 1

The following program $\mathcal{P}$ defines the sorting, in increasing order, of a list by straight insertion:

```
1.  SORT([], []).
2.  SORT([a|y], x) :- SORT(y, z), INS(a, z, x).
3.  INS(a, [], [a]).
4.  INS(a, [b|t], [b|s]) :- (a>b), INS(a, t, s).
5.  INS(a, [b|t], [a|[b|t]]) :- (a≤b).
```

A simple specification for $\mathcal{P}$ is:

$SORT_{\mathcal{S}}(x, y) \leftrightarrow \mathcal{S}_{SORT}(x, y)$

where $\mathcal{S}_{SORT}(x, y) =_{def}$ nlist(x) $\wedge$ nlist(y) $\wedge$ perm(x, y) $\wedge$ incr(y)

$INS_{\mathcal{S}}(a, z, x) \leftrightarrow \mathcal{S}_{INS}(a, z, x)$

where $\mathcal{S}_{INS}(a, z, x) =_{def}$ nat(a) $\wedge$ nlist(z) $\wedge$ nlist(x) $\wedge$

$\exists x_1, x_2.\ (x = x_1 \bullet [a|x_2]) \wedge (z = x_1 \bullet x_2) \wedge$ greater(a, x_1) $\wedge$ minor(a, x_2)

In $\mathcal{S}_0$ the following predicates are defined:

nlist(x) $=_{def}$ x=[] $\vee$ ($\exists$a, y. nat(a) $\wedge$ nlist(y) $\wedge$ x=[a|y]) ;

nat(a) $=_{def}$ x=0 $\vee$ ($\exists$y. nat(y) $\wedge$ (x=y+1)) ;

perm(x, y) $=_{def}$ $\forall$a. occ(a, x) = occ(a, y) ,

with occ(e, t) integer function which counts the occurrences of the element e in the list t ;

incr(y) $=_{def}$ (y = []) $\vee$ ($\exists$b. y = [b]) $\vee$ ($\forall$b, c. precede(b, c, y) $\rightarrow$ (b $\leq$ c)) ,

with precede(b, c, y) $=_{def}$ $\exists x_1, x_2, x_3.\ y = x_1 \bullet [b|x_2] \bullet [c|x_3]$;

greater(a, x) $=_{def}$ $\forall$b. member(b, x) $\rightarrow$ (a>b) ;

smaller-eq(a, x) $=_{def}$ $\forall$b. member(b, x) $\rightarrow$ (a$\leq$b) ;

minor(a, x_2) $=_{def}$ ($x_2 \neq []$) $\rightarrow$ (a $\leq$ car(x_2)) ,

where car , $\bullet$ (append) and member are the usual operators on lists.

Thus $\mathcal{S}$ extends a theory of natural numbers and lists of natural numbers, $\mathcal{S}_0$, with the two axioms:

$SORT_{\mathcal{S}}(x, y) \leftrightarrow \mathcal{S}_{SORT}(x, y)$

$INS_{\mathcal{S}}(a, z, x) \leftrightarrow \mathcal{S}_{INS}(a, z, x)$.

In order to prove partial correctness of $\mathcal{P}$ with respect to $\mathcal{S}$, it is sufficient to prove that for each clause the corresponding formula in $\mathcal{S}$ is a theorem:

1. $\mathcal{S}$ |– $SORT_{\mathcal{S}}([], [])$
2. $\mathcal{S}$ |– ($SORT_{\mathcal{S}}(y, z) \wedge INS_{\mathcal{S}}(a, z, x)) \rightarrow SORT_{\mathcal{S}}([a|y], x)$
3. $\mathcal{S}$ |- $INS_{\mathcal{S}}(a, [], [a])$
4. $\mathcal{S}$ |- ((a>b) $\wedge$ $INS_{\mathcal{S}}(a, t, s)) \rightarrow INS_{\mathcal{S}}(a, [b|t], [b|s])$
5. $\mathcal{S}$ |- (a $\leq$ b) $\rightarrow$ $INS_{\mathcal{S}}(a, [b|t], [a|[b|t]])$

The proofs depend on some properties which hold in $\mathcal{S}_0$ such as:

$\forall$a,z,x,y. (perm([a|z], x) $\wedge$ perm(y, z)) $\rightarrow$ perm([a|y], x)

and on the following lemmas which hold in $\mathcal{S}$:

$\forall$a,z,x. $INS_{\mathcal{S}}(a, z, x) \rightarrow$ perm([a|z], x)

$\forall$a,z,x. (incr(z) $\wedge$ $INS_{\mathcal{S}}(a, z, x)) \rightarrow$ incr(x).

For details see [Bos88].

2.2. Completeness

In order to deal with completeness of a program, we find it convenient to compare the theory $\mathcal{S}$, which describes the intended meaning of $\mathcal{P}$, with the theory Comp($\mathcal{P}$), the completion of $\mathcal{P}$. Let us recall how the completion of $\mathcal{P}$ [Llo84] is obtained.

The first step is to transform each clause c

c: $\forall\ y_1, \ldots, y_h.\ (p(t_1, \ldots, t_k) \leftarrow A_1 \wedge \ldots \wedge A_m)$, $m \geq 0$,

into the general form

$p(x_1, \ldots, x_k) \leftarrow \exists\, y_1, \ldots, y_h\ (x_1=t_1 \wedge \ldots \wedge x_k=t_k \wedge A_1 \wedge \ldots \wedge A_m)$

where $x_1, \ldots, x_k$ are new variables.

Let

$p(x_1, \ldots, x_k) \leftarrow E_1$

:

$p(x_1, \ldots, x_k) \leftarrow E_j$

be the general form of the finitely many clauses about p in $\mathcal{P}$.

The completed definition of p in $\mathcal{P}$ is

$p(x_1, \ldots, x_k) \leftrightarrow E_1 \vee \ldots \vee E_j$.

The completion of $\mathcal{P}$, Comp($\mathcal{P}$), contains all the completed definitions of the predicate symbols of $\mathcal{P}$, together with an axiomatization of the equality theory.

It is a well known result [Llo84] that the positive facts deducible from Comp($\mathcal{P}$) are exactly the ones deducible from $\mathcal{P}$, that is

$\forall p.\ \mathcal{M}(p)_{\mathcal{P}} = \mathcal{M}(p)_{Comp(\mathcal{P})}$.

Hence the completeness of $\mathcal{P}$ with respect to a specification $\mathcal{S}$ can be expressed as:

$\forall p.\ \mathcal{M}(p)_{Comp(\mathcal{P})} \supseteq \mathcal{M}(p)_{\mathcal{S}}$

which means to prove:

for any k-ary predicate symbol p in $\mathcal{P}$ and any tuple $(t_1, \ldots, t_k)$,

if $\mathcal{S} \vdash p_{\mathcal{S}}(t_1, \ldots, t_k)$ then $Comp(\mathcal{P}) \vdash p(t_1, \ldots, t_k)$.

Let p_1 of arity k_1, ... , p_n of arity k_n be the predicate symbols in $\mathcal{P}$, then in order to prove the completeness of $\mathcal{P}$ with respect to $\mathcal{S}$ it is sufficient to prove, in (Comp($\mathcal{P}$)+$\mathcal{S}$) ; that:

$(\forall\, x_{1,1}, \ldots, x_{1,k1}.\ p_{1\mathcal{S}}(x_{1,1}, \ldots, x_{1,k1}) \rightarrow p_1(x_{1,1}, \ldots, x_{1,k1})) \wedge \ldots \wedge$ (2)

$(\forall\, x_{n,1}, \ldots, x_{n,kn}.\ p_{n\mathcal{S}}(x_{n,1}, \ldots, x_{n,kn}) \rightarrow p_n(x_{n,1}, \ldots, x_{n,kn}))$.

Note that the correctness of this condition depends on the way the specification has been defined:

i) the languages of $\mathcal{P}$ and $\mathcal{S}$ have in common only predicate symbols in the theory of data domains: $L_{\mathcal{S}} \cap L_{\mathcal{P}} = L_{\mathcal{S}0}$;

ii) $\mathcal{S}$ is a conservative extension of $\mathcal{S}_0$ since only the definitions of the new predicate symbols associated to the ones in $\mathcal{P}$ have been added.

Example 2

Let $\mathcal{P}$ be

greater_equal(x,0).

greater_equal(x+1,y+1) :- greater_equal(x,y).

Let $\mathcal{S}$ be a theory of natural numbers augmented with the definition:

$\mathcal{S}_{greater\_equal}(x,y) =_{def} nat(x) \wedge nat(y) \wedge \exists z.\ (nat(z) \wedge x=y+z)$

and the axiom:

$greater\_equal_{\mathcal{S}}(x,y) \leftrightarrow \mathcal{S}_{greater\_equal}(x,y)$.

(Comp($\mathcal{P}$)+$\mathcal{S}$) contains both the previous axiom and

$greater\_equal(x,y) \leftrightarrow (y=0) \vee (\exists x_1,y_1.\ x=x_1+1 \wedge y=y_1+1 \wedge greater\_equal(x_1,y_1))$

plus the theory of natural numbers $\mathcal{S}_0$.

Then, for the completeness of $\mathcal{P}$ with respect to $\mathcal{S}$, it is sufficient to prove that

$\forall x,y.\ greater\_equal_{\mathcal{S}}(x,y) \rightarrow greater\_equal(x,y)$

is a theorem in (Comp($\mathcal{P}$)+$\mathcal{S}$).

Note that the completed definition of $p_1, \ldots, p_n$ in Comp($\mathcal{P}$) may be (possibly mutually) recursive:

$p_1(\underline{x}_1) \leftrightarrow C_1(p_1, \ldots, p_n)(\underline{x}_1)$

:

$p_n(\underline{x}_n) \leftrightarrow C_n(p_1, \ldots, p_n)(\underline{x}_n)$,

where $\underline{x}_i$ is an abbreviation for $x_{i,1}, \ldots, x_{i,ki}$ and $C_i(p_1, \ldots, p_n)(\underline{x}_i)$ is a disjunction of existential quantifications of conjunctions: $E_1 \vee \ldots \vee E_{hi}$, with E_l, $1 \leq l \leq h_i$, in the form $\exists \underline{y}. (B_1 \wedge \ldots \wedge B_m)$, where $B_1, \ldots, B_m$ are equalities or atomic predicates $p_i(\underline{t})$, $1 \leq i \leq n$.

Let us consider a simple case, when there exists an ordering of the predicate symbols in $\mathcal{P}$ such that, for any i, $1 \leq i \leq n$, the definition of p_i does not depend on $p_{i+1}, \ldots, p_n$

$p_1(\underline{x}_1) \leftrightarrow C_1(p_1)(\underline{x}_1)$

:

$p_i(\underline{x}_i) \leftrightarrow C_i(p_1, \ldots, p_i)(\underline{x}_i)$

:

$p_n(\underline{x}_n) \leftrightarrow C_n(p_1, \ldots, p_n)(\underline{x}_n)$.

Then we can separately prove each implication by following the order on the predicate symbols

$\forall \underline{x}_i.\ p_{i\mathcal{S}}(\underline{x}_i) \rightarrow p_i(\underline{x}_i)$, $1 \leq i \leq n$,

that is

$$\forall \underline{x}_i.\ p_{i\mathcal{S}}(\underline{x}_i) \rightarrow C_i(p_1, \ldots, p_i)(\underline{x}_i) \ , \quad 1 \leq i \leq n \ . \tag{3}$$

In order to prove the i-th implication in (3), we can use the fact that the j-th implications hold for any $j < i$. This, for the structure of the formula C_i, guarantees that

$\forall \underline{x}_i.\ C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{x}_i) \rightarrow C_i(p_1, \ldots, p_i)(\underline{x}_i)$.

Then, in order to prove (3), it is sufficient to prove, in (Comp($\mathcal{P}$)+$\mathcal{S}$)

$$\forall \underline{x}_i.\ p_{i\mathcal{S}}(\underline{x}_i) \rightarrow C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{x}_i) \ , \quad 1 \leq i \leq n \ . \tag{4}$$

Since we assumed that each variable $x_{i,j}$ is typed, whenever we have well-ordering relations on the types associated to the universally quantified variables (i.e. we are dealing with well-founded sets), we can use structural induction in the proof of (4), by defining an extension of these orderings onto n-tuples. Note that, in proving the inductive step

$p_{i\mathcal{S}}(\underline{t}) \rightarrow C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{t})$,

we can use the inductive hypothesis $p_{i\mathcal{S}}(\underline{w}) \rightarrow C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{w})$, for $\underline{w} < \underline{t}$.

Since, as we said before,

$\forall \underline{x}_i.\ C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{x}_i) \rightarrow C_i(p_1, \ldots, p_i)(\underline{x}_i)$ and $\forall \underline{x}_i.\ C_i(p_1, \ldots, p_i)(\underline{x}_i) \rightarrow p_i(\underline{x}_i)$,

this means that $p_{i\mathcal{S}}(\underline{w}) \rightarrow p_i(\underline{w})$ holds for any $\underline{w} < \underline{t}$. Then again, for the structure of the formula C_i, it might be sufficient to prove in (Comp($\mathcal{P}$)+$\mathcal{S}$):

$p_{i\mathcal{S}}(\underline{t}) \rightarrow C'_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_{i\mathcal{S}}, p_i)(\underline{t})$

where $C'_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_{i\mathcal{S}}, p_i)(\underline{t})$ is obtained by substituting $p_{i\mathcal{S}}(\underline{w})$ to $p_i(\underline{w})$, for all $\underline{w} < \underline{t}$, in $C_i(p_{1\mathcal{S}}, \ldots, p_{i-1\mathcal{S}}, p_i)(\underline{t})$. This is particularly useful whenever $\mathcal{S}$ does not contain recursive definitions as shown in the examples.

Example 3

In the second example, for the completeness of $\mathcal{P}$ with respect to $\mathcal{S}$, we have to prove:

$\forall$ x,y. (($nat(x) \wedge nat(y) \wedge \exists z.\ (nat(z) \wedge x=y+z)$)

$\rightarrow$ ($y=0 \vee (\exists x_1,y_1,z.\ x=x_1+1 \wedge y=y_1+1 \wedge greater\_equal(x_1,y_1))$).

The proof is very simple: it can be given by induction on the couples (x, y) with the ordering

$(x,y) < (x',y')$ iff $x+y < x'+y'$.

For details see [Bos88].

Example 4

In order to prove the completeness of the sorting program, $\mathcal{P}$, of the first example, with respect to the specification $\mathcal{S}$, we show, for any p_i in $\mathcal{P}$,

$(Comp(\mathcal{P})+\mathcal{S}) \vdash \forall \underline{x}\ .\ p_{i\mathcal{S}}(\underline{x}) \rightarrow p_i(\underline{x})$.

In Comp($\mathcal{P}$) there are the definitions:

$SORT(u,v) \leftrightarrow \mathcal{C}_{SORT}(u,v)$

with $\mathcal{C}_{SORT}(u,v) =_{def} (u=[] \wedge v=[]) \vee (\exists a,y,z.\ u=[a|y] \wedge SORT(y, z) \wedge INS(a, z, v))$

$INS(a,z,x) \leftrightarrow \mathcal{C}_{INS}(a,z,x)$

with $\mathcal{C}_{INS}(a,z,x) =_{def}$ $(z=[] \wedge x=[a])$

$\vee (\exists b,t,s.\ z=[b|t] \wedge x=[b|s] \wedge a>b \wedge INS(a,t, s))$

$\vee (\exists b,t.\ z=[b|t] \wedge x=[a|[b|t]] \wedge a \leq b)$

Since the axioms defining SORT and INS are not mutually recursive, we can first prove the completeness of INS, which does not depend on SORT.

1. $\forall$ a,z,x. $INS_{\mathcal{S}}(a,z,x) \rightarrow INS(a,z,x)$

We have to prove that

$\forall$ a,z,x. $nat(a) \wedge nlist(z) \wedge nlist(x)$

$\wedge (\exists x_1, x_2.\ (x = x_1 \bullet [a|x_2]) \wedge (z = x_1 \bullet x_2) \wedge greater(a, x_1) \wedge minor(a, x_2))$

$\rightarrow$ $(z=[] \wedge x=[a])$

$\vee (\exists b,t,s.\ z=[b|t] \wedge x=[b|s] \wedge a>b \wedge INS(a,t, s))$

$\vee (\exists b,t.\ z=[b|t] \wedge x=[a|[b|t]] \wedge a \leq b)$.

The proof can be done by structural induction on z, for details see [Bos88].

2. $\forall$ u,v. $SORT_{\mathcal{S}}(u,v) \rightarrow SORT(u,v)$

Since we already proved $\forall$ a,z,x. $INS_{\mathcal{S}}(a,z,x) \rightarrow INS(a,z,x)$, we have to prove that

$\forall u,v\ (nlist(u) \wedge nlist(v) \wedge perm(u, v) \wedge incr(v))$

$\rightarrow$ $(u=[] \wedge v=[]) \vee (\exists a,y,z.\ u=[a|y] \wedge SORT(y, z) \wedge INS_{\mathcal{S}}(a, z, v))$

The proof can be by structural induction on the pair of lists (u,v) with the lexicographic ordering:

$(u, v) < (u', v')$ iff $length(u) < length(u')$ or

$length(u) = length(u')$ and $length(v) < length(v')$.

For details see [Bos88].

When the completed definitions in Comp($\mathcal{P}$) are strictly mutually recursive, we can transform (2) into

$\forall \underline{x}_1, \ldots, \underline{x}_k.\ (p_{1\mathcal{S}}(\underline{x}_1) \rightarrow p_1(\underline{x}_1) \wedge \ldots \wedge p_{k\mathcal{S}}(\underline{x}_k) \rightarrow p_k(\underline{x}_k))$ (5)

by a suitable renaming of variables.

Then we can prove (5) by a global structural induction, i.e., when the tuple $\underline{t}_1, \ldots, \underline{t}_k$ is considered, in proving

$(p_{1\mathcal{S}}(\underline{t}_1) \rightarrow p_1(\underline{t}_1)) \wedge \ldots \wedge (p_{k\mathcal{S}}(\underline{t}_k) \rightarrow p_k(\underline{t}_k))$

we are allowed to assume (inductive hypothesis)

$(p_{1\mathcal{S}}(\underline{w}_1) \rightarrow p_1(\underline{w}_1)) \wedge \ldots \wedge (p_{k\mathcal{S}}(\underline{w}_k) \rightarrow p_k(\underline{w}_k))$, for any $(\underline{w}_1, \ldots, \underline{w}_k) < (\underline{t}_1, \ldots, \underline{t}_k)$.

In order to illustrate this case, we add a further simple example.

Example 5

Let the program $\mathcal{P}$ contain the following clauses, which can be used to determine if the length of a list is even or odd:

1. PAIRL([]).
2. PAIRL([a|x]) :- ODDL(x).
3. ODDL([a|x]) :- PAIRL(x).

Comp($\mathcal{P}$) contains:

$PAIRL(y) \leftrightarrow (y = [] \vee (\exists\, a,x.\ y = [a|x] \wedge ODDL(x)))$

$ODDL(y) \leftrightarrow (\exists\, a,x.\ y = [a|x] \wedge PAIRL(x))$.

We want to prove the completeness of $\mathcal{P}$ with respect to the specification:

$\mathcal{S}_{PAIRL}(y) =_{def}$ length(y) $\equiv 0 \bmod 2$

$\mathcal{S}_{ODDL}(y) =_{def}$ length(y) $\equiv 1 \bmod 2$

where length([]) = 0

length([a|x]) = length(x) +1.

In order to prove completeness it is sufficient to prove that:

$\forall\, x,y.\ ((PAIRL_{\mathcal{S}}(x) \rightarrow PAIRL(x)) \wedge (ODDL_{\mathcal{S}}(y) \rightarrow ODDL(y)))$

holds in (Comp($\mathcal{P}$)+$\mathcal{S}$). This can be proved by induction on (x,y) by choosing the lexicographic ordering, see [Bos88] for a detailed proof.

3. Verifying properties of program modules

In this section we will generalize our discussion on the correctness of logic programs with respect to their specification by considering also module specifications.

Modularity of programs comes very natural in top-down design since modularization is the first step in stepwise refinement. Modularity contributes to increase correctness and moreover allows for reusability of programs. In fact program modules correspond to abstract data types and operations and can be used in different applications. In order to exploit the advantages of modularization, each module has a specification associated to it. This is necessary both for describing the decomposition of the problem at design level and for reusing the software. Thus, once more, verifying the correct implementation of a module with respect to its specification becomes necessary. Often the specification describes only some properties of the program module, the ones we are actually interested in. This is also true when we want to characterize the module's behavior for a particular data domain, as it happens in program specialization and optimization where one is interested in the equivalence of program modules on a restricted domain. Therefore, module specification allows for both correctness and efficiency improvement.

In general a module specification has the following form

{Pre($\underline{x}$)} M($\underline{x}$) {Post($\underline{x}$)}

where $\underline{x}$ is a tuple of variables, Pre($\underline{x}$) is a precondition , that is a predicate describing the context of execution of the module M($\underline{x}$), and Post($\underline{x}$) is a postcondition, that is a predicate describing the context after

the execution of M(x). In procedural programming pre- and post-conditions describe the state of computation before and after the execution of the module.

In logic programming modularity is forced by the programming style. A module is given by the set of definite clauses defining a predicate symbol. This intrinsic modularity can be, in our opinion, an advantage of logic programming over conventional programming but, in order to achieve all the advantageous consequences of it (clarity, correctness, reusability, possibility of program transformations), it is necessary to associate specifications to modules. This is obviously not necessary (and rather unrealistic) for small modules as the ones defining base predicates. The modular structure of logic programs is an advantage only for high level programming, when we can control it. It would be rather unpractical to have to specify all the predicates in the program! In top-down design the problem is decomposed into subproblems. Each subspecification is implemented by a program module. The process is repeated until useful, that is until specifications of subproblems are necessary in order to abstract from implementation details. When this is not the case, that is for simple predicates, the declarative semantics can be used as specification for the module. Then what is the meaning of

$\{Pre(\underline{x})\}\ M(\underline{x})\ \{Post(\underline{x})\}$

where $M(\underline{x})$ [1] is defined by a set of clauses in a logic program $\mathcal{P}$?

In the declarative semantics relations are associated to predicates. Hence $Pre(\underline{x})$ describes the relation to which $M(\underline{x})$ is applied, the world from which the tuples satisfying $M(\underline{x})$ are taken, we could call it the initial relation . Analogously $Post(\underline{x})$ indicates the final relation , that is the one produced from $Pre(\underline{x})$ by $M(\underline{x})$. Thus a logic program module $M(\underline{x})$ is partially correct with respect to a pre/post specification if and only if it defines a subrelation of the precondition which is included in the postcondition:

$\{Pre(\underline{x})\}\ M(\underline{x})\ \{Post(\underline{x})\}$ is partially correct iff

$Post(\underline{x}) \supseteq (Pre(\underline{x}) \cap M(\underline{x}))$.

In order to prove partial correctness of $M(\underline{x})$ with respect to $Pre(\underline{x})/Post(\underline{x})$, we will have to prove in ($\mathfrak{S}_0$ + Comp($\mathcal{P}$)):

$$\forall \underline{x} . (Pre(\underline{x}) \wedge M(\underline{x})) \rightarrow Post(\underline{x}). \qquad (1)$$

Thus a program module $M(\underline{x})$, partially correct with respect to a pre/post specification, could be thought of as a filter which, when correctly used, that is when $Pre(\underline{x})$ is satisfied, selects only tuples which satisfy $Post(\underline{x})$.

If the precondition is true, there is no initial restriction on the world. Thus $M(\underline{x})$ is partially correct with respect to the specification given from a postcondition if and only if the relation defined by $M(\underline{x})$ is included in the postcondition:

$M(\underline{x})\ \{Post(\underline{x})\}$ is partially correct iff

$Post(\underline{x}) \supseteq M(\underline{x})$

that is, in order to prove partial correctness of $M(\underline{x})$ with respect to $Post(\underline{x})$, we will have to prove in ($\mathfrak{S}_0$ + Comp($\mathcal{P}$)):

$$\forall \underline{x} . M(\underline{x}) \rightarrow Post(\underline{x}) . \qquad (2)$$

Obviously any program module is partially correct with respect to the postcondition true.

---

(1) Note that in the following we will indicate with $M(\underline{x})$ both the set of clauses defining the predicate symbol and the predicate itself.

A logic program module M(x) is totally correct with respect to a pre/post specification if and only if the relation filtered by the module is the final one:

{Pre(x)} M(x) {Post(x)} is totally correct iff

Post(x) = (Pre(x) ∩ M(x))

that is, in order to prove total correctness of M(x) with respect to Pre(x)/Post(x), we will have to prove in ($\mathcal{S}_0$ + Comp($\mathcal{P}$)):

$\forall \underline{x} . (Pre(\underline{x}) \wedge M(\underline{x})) \leftrightarrow Post(\underline{x})$. (3)

Note that a logic program module can be partially (totally) correct with respect to many different pre/post specifications. These correspond to different ways of using the module, for example to different input/output functionalities. Moreover a program module M(x), with precondition Pre(x), can be partially correct with respect to a postcondition, Post1(x) , and at the same time totally correct with respect to another postcondition, Post2(x). This means that the second postcondition is included in the first:

if

{Pre(x)} M(x) {Post1(x)} is partially correct and

{Pre(x)} M(x) {Post2(x)} is totally correct

then

Post1(x) ⊇ Post2(x) .

If there is no precondition (Pre(x) = true), we will call Post2(x) strongest specification of M(x) and Post1(x) weak specification of M(x). Strongest specifications determine a unique relation, the one associated to the declarative semantics of the program module M(x).

The assertion

M(x) {Post(x)}

interpreted as a strongest specification of the module, uniquely individuates the class of equivalent program modules for which

M(x) = Post(x) ,

that is the class of programs which are totally correct with respect to the specification Post(x).

On the other hand, as we already pointed out, a given module M(x) can be partially correct with respect to many different weak specifications. The strongest specification of a program module M(x) is in fact the intersection of all the weak specifications of the module. Weak specifications can be used for describing program properties.

If a module M(x) is partially correct with respect to a pre/post specification

{Pre(x)} M(x) {Post(x)}

we know that any use of it which satisfies the precondition, will satisfy the postcondition too. In other terms, given a query

? M(t) , with t tuple of terms ,

if Pre(t) is satisfied, then for any successful answer substitution σ, the postcondition Post(t σ) holds:

if Pre(t) then

for any substitution σ , M(t σ) → Post(t σ) .

Hence if a module M(x) is partially correct with respect to a pre/post specification with precondition true, this means that any use of it, that is any query ? M(t), with t tuple of terms, is correct and allowed. For each tuple which satisfies the query, the postcondition is guaranteed to hold.

To sum up, the pre/post specification of a logic module, which is partially correct with respect to it, gives us information about properties of successful computations for queries which satisfy the precondition. When the query does not satisfy the precondition, we can not infer anything from the specification of the module. This is exactly analogous to what happens in procedural programming. There in fact a pre/post specification describes the behavior of a program if it starts the execution in a state which satisfies the precondition. Compared with procedural or functional programs, a logic program is much more general since it allows for multiple input/output readings and generic uninstantiated answers. This is the reason why many pre/post specifications are associated to a logic program module.

Proving properties of programs generally consists in proving their partial correctness with respect to some pre/post specification. We give now a sufficient criterion for verifying partial correctness of a program module with respect to a pre/post specification. We assume the program module to be asserted, which means that all the predicates in it have an associated pre/post specification. For simple predicates we consider their declarative semantics as the associated specification. In fact, as we said before, each predicate, $p(\underline{x})$, is totally correct with respect to the specification

$p(\underline{x})$ $\{Comp_{p(\underline{x})}\}$

where $Comp_{p(\underline{x})}$ is the completed definition of p in the program $\mathcal{P}$ and it supplies the declarative semantics of p.

The intuitive idea behind the sufficient criterion is to consider separately each clause in the module definition in analogy to the sufficient criterion for partial correctness in section 2. We assume that the precondition is satisfied and we prove that, in this hypothesis, also the postcondition is satisfied. We use an induction on the number of derivation steps of the predicate $M(\underline{x})$, while predicates invoked inside the module definition, that is in the antecedent of the clause, are assumed to be partially correct with respect to their pre/post specifications.

Sufficient Criterion (4)

Let $M(\underline{x})$ be an asserted module

$\{Pre(\underline{x})\}$ $M(\underline{x})$ $\{Post(\underline{x})\}$

in a program $\mathcal{P}$ and $\mathcal{S}_0$ the theory which describes the data domains, then $M(\underline{x})$ is partially correct with respect to its pre/post specification if the following conditions hold in $(\mathcal{S}_0 + Comp(\mathcal{P}))$:

a) each module invoked by $M(\underline{x})$, that is each module corresponding to predicate symbols in the body of $M(\underline{x})$ clauses, is partially correct with respect to its specification;

b) for each fact, $M(\underline{t})$,
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \rightarrow Post(\underline{t})$, $x_1,\ldots,x_n$ variables in $\underline{t}$;

c) for each clause, $M(\underline{t})$:- $A_1(\underline{t}_1), \ldots, A_n(\underline{t}_n)$., in $M(\underline{x})$,
if $\{P_i(\underline{y}_i)\}$ $A_i(\underline{y}_i)$ $\{Q_i(\underline{y}_i)\}$ are the specifications of the predicates in the body of the clause, then
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \rightarrow P_1(\underline{t}_1)$
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \wedge Q_1(\underline{t}_1) \rightarrow P_2(\underline{t}_2)$
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \wedge Q_1(\underline{t}_1) \wedge Q_2(\underline{t}_2) \rightarrow P_3(\underline{t}_3)$
. . .
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \wedge Q_1(\underline{t}_1) \wedge \ldots \wedge Q_{n-1}(\underline{t}_{n-1}) \rightarrow P_n(\underline{t}_n)$
$\forall x_1,\ldots,x_n.\ Pre(\underline{t}) \wedge Q_1(\underline{t}_1) \wedge \ldots \wedge Q_n(\underline{t}_n) \rightarrow Post(\underline{t})$.

The advantage of this sufficient criterion consists in its modularity since, by considering separately each logic module and each clause in a module definition, it partitions the proof of M(x) into many small ones.

Theorem (soundness of the sufficient criterion for partial correctness (4)).

Let M(x) be an asserted logic module, {Pre(x)} M(x) {Post(x)}, in a program $\mathcal{P}$, if the conditions (a), (b) and (c) in (4) hold, then M(x) is partially correct with respect to its pre/post specification, that is:

$\forall \underline{x}\,.\,(\mathrm{Pre}(\underline{x}) \wedge \mathrm{M}(\underline{x})) \rightarrow \mathrm{Post}(\underline{x})$.

We omit the proof, which can be found in [Bos88].

In the following we give a very simple example of property verification on a logic program module.

Example 6

The program module is taken from the first example in the previous section:

1: ins(a, [], [a]).

2: ins(a, [b|u], [b|v]) :- (a>b), ins(a, u, v).

3: ins(a, [b|u], [a|[b|u]]) :- (a≤b).

We recall that $\mathbf{S}_0$ is a theory of natural numbers and lists of natural numbers and that the meaning of ins(r, z, x) is that r is a natural number, z and x are lists of natural numbers and x is equal to the ordered insertion of r into z.

The property we are interested in, is the fact that, if z is in increasing order, then also x is in increasing order. This can be represented by partial correctness with respect to the specification

{incr(z)} ins(r, z, x) {incr(x) ∧ perm([r|z], x)}

where the definitions

incr(z) $=_{\mathrm{def}}$ (z = []) ∨ (∃b. z = [b]) ∨ (∀b, c. precede(b, c, z) → (b ≤ c))

precede(b, c, z) $=_{\mathrm{def}}$ $\exists x_1, x_2, x_3.\ z = x_1 \bullet [b|x_2] \bullet [c|x_3]$;

perm(x, y) $=_{\mathrm{def}}$ ∀a. occ(a, x) = occ(a, y)

are in $\mathbf{S}_0$. We have omitted the obvious typing of parameters.

The specifications of the predicates in the body of the clauses are the obvious ones:

{true} (x>y) {x>y}

{true} (x≤y) {x≤y},

then the sufficient criterion (4) tells us that in order to prove partial correctness of the program module with respect to its pre/post specification it is sufficient to prove in $\mathbf{S}_0$:

a) each submodule is partially correct with respect to its specification: this is trivially true since the predicates in the body of the clauses are base predicates specified with their declarative semantics;

b) 1) - **incr([]) → (incr([a]) ∧ perm([a], [a]))**;

c) 2) - **incr([b|u]) → true**;

- **(incr([b|u]) ∧ (a>b)) → incr(u)**;

- **(incr([b|u]) ∧ (a>b) ∧ incr(v) ∧ perm([a|u], v))**
  → ((incr([b|v]) ∧ perm([a|[b|u]], [b|v]));

3) - **incr([b|u]) → true**;

- **(incr([b|u]) ∧ true ∧ (a≤b)) → (incr([a|[b|u]]) ∧ perm([a|[b|u]], [a|[b|u]]))**.

The proofs, which as usual depend on properties which hold in $\mathbf{S}_0$, can be found in [Bos88].

In this way we prove that any query with the second parameter in increasing order will produce a third parameter also in increasing order. Nothing can be said for the following queries:

? ins(3, [1,5,2,4], y)

? ins(2, x, y)

? ins(10, y, [1,2,5,10])

? ins(x, y, [1,7,5]) .

In this example the precondition implicitly defines a mode: only queries with the second parameter ground and in increasing order are described by this pre/post specification. This does not always happen, for example the pre/post specification

{x=y} P(x,y) {Post(x,y)}

does not imply any mode, it only imposes that the query has two equal parameters. All the following queries satisfy the precondition

? P(100, 100)

? P([1,4,3,5], [1,4,3,5])

? P([1|x], [1|x])

? P(z, z)

? P([s|t], [s|t]) .

Our sufficient criterion for partial correctness of a logic program module with respect to a pre/post specification is similar to Drabent and Maluszynski's inductive assertion method [Dra87]. Such a method was proposed for verifying run-time properties of logic programs, but it could be modified in order to use it for general properties of declarative semantics. In fact it is a translation of the procedural verification method "a'la Hoare" into the logic programming framework. For declarative semantics it becomes just a sequential, left-to-right, composition of procedure calls without side-effects. In [Bos88] we show how to modify their method in order to make it applicable exactly to the same context.

4. Conclusions

The major contribution of this paper consists in showing modular techniques for proving properties of logic programs. We gave a sufficient criterion for the completeness of a logic program with respect to a given specification and one for partial correctness with respect to more general properties expressed in terms of pre/post specifications. Verifying program correctness is not always an easy task, namely for large, complicated programs. But we believe that, as it happens in more traditional programming, the insight one gets from this detailed analysis is surely worth the effort.

References

[Apt82] K.R. Apt, M.H. van Emden, *Contributions to the theory of Logic Programming*, JACM 29, N.23 (1982) pp. 841-862.

[Bal78] K. Balogh, *On an Interactive Program Verifier for Prolog Programs*, Colloquia Mathematica Societatis J. Bolyai 26 - Mathematical Logic in Computer Science - Hungary (1978).

[Bos87] A. Bossi, N. Cocco, S. Dulli, *A Method for Specializing Logic Programs*, (1987) submitted for pubblication.

[Bos88] A. Bossi, N. Cocco, *On the correctness of Logic Programs*, Rapporto Interno del Dip. Matematica Pura ed Applicata, Università di Padova (1988).

[Bur77] R. M. Burstall, J. Darlington, *A Transformation System for developing Recursive Programs*, JACM 24, N.1 (1977) pp. 44-67.

[Cla77] K.L. Clark, S. Tarnlund, *A first order theory of data and programs*, Proc. IFIP 77, (1977) pp 939-944.

[Dra87] W. Drabent, J. Maluszynski, *Inductive assertion Method for logic programs*, TAPSOFT 87 , LNCS 250, (1987) pp. 167-181.

[vanE76] M.H. van Emden, R.A. Kowalski, *The Semantics of Predicate Logic as a Programming Language*, JACM 23, N.4 (1976) pp.733-742.

[Flo67] R. W. Floyd, *Assigning meanings to programs*, Proc. A.M.S. Symp. in Applied Mathematics, Amer. Math. Soc. (1967) pp. 19-31.

[Fra85] N. Francez, O. Grumberg, S. Katz and A. Pnueli, *Proving Termination of Prolog Programs*, Proc. Logics of Programs, LNCS 193, (1985) pp. 89-105.

[Fut87] Y. Futamura, K. Nogi, *Generalized Partial Computation*, IFIP TC2 Work. Conf. on Partial and Mixed Computation, Denmark (1987).

[Gal86] J. Gallagher, *Transforming Logic Programs by Specializing Interpreters*, Proc. ECAI 86 (1986).

[Hoa69] C. A. R. Hoare, *An axiomatic basis for computer programming*, CACM 12, (1969) pp. 576-583.

[Hog81] C.J. Hogger, *Derivation of Logic Programs*, JACM 29, N.2 (1981) pp. 372-392.

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

[Llo84] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag 1984.

[Pra65] D. Prawitz, *Natural Deduction: A Proof-Theoretical Study*, Almqvist & Wiksell, Stockholm, 1965.

[Sato84] T.Sato, H. Tamaki, *Transformational Logic program Synthesis*, Proc. of the Int. Conf. on Fifth Generation Computer Systems 1984, ICOT 1984.

[Tam84] H. Tamaki, T. Sato, *Unfold/fold transformation of logic programs,* Proc. II International Logic Programming Conference, Upssala (1984) pp. 127-138.