PROOFS OF DECLARATIVE PROPERTIES OF LOGIC PROGRAMS

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cédex
Tel. : (33-1) 39 63 55 36
uucp: deransar@minos.inria.fr

Abstract

In this paper we shall consider proofs of declarative properties of Logic Programs, i.e. properties associated with the logical semantics of pure Logic Programs, in particular what is called the partial correctness of a logic program with respect to a specification. A specification consists of a logical formula associated with each predicate and establishing a relation between its arguments. A definite clause program is partially correct iff every possible answer substitution satisfies the specification.

This paper generalizes known results in logic programming in two ways : first it considers any kind of specification, second its results can be applied to extensions of logic programming such as functions or constraints.

In this paper we present two proof methods adapted from the Attribute Grammar field to the field of Logic Programming. Both are proven sound and complete. The first one consists of defining a specification stronger than the original one, which furthermore is inductive (fix-point induction).

The second method is a refinement of the first one : with every predicate, we associate a finite set of formulas (we call this an annotation), together with implications between formulas. The proofs become more modular and tractable, but the user has to verify the consistency of his proof, which is a decidable property. This method is particularly suitable for proving the validity of specifications which are not inductive.

Introduction

The problem of proving the correctness of a definite Horn clause program (clauses with exactly one positive literal or logic programs), with respect to a given specification, saying what the answer substitutions (if any) should be, although essential, has been rarely considered [Cla 79, Hog 84, Dev 87, DrM87, SS 86]. Most published works present some variant of the inductive assertion method (fix-point induction) which is illustrated with examples. No adaptation of the presentation to the field of logic programming is always made, no other original method is presented and no proof of soundness and completeness of the method are always given.

In this paper we shall consider the partial correctness of a logic program with respect to a specification. A specification consists of a logical formula associated with each predicate and establishing a relation between its arguments. A definite clause program is partially correct iff every possible answer substitution satisfies the specification.

Partial correctness does not depend on any interpretation strategy ; it does not refer to any operational semantics of logic programs but only to their constructive (proof-trees) or logical semantics [DF 88]. Specifications are properties of the proof-tree roots. They are used to detect properties of goals before any execution or testing of a logic program.

The proof methods presented in this paper are part of a general methodology we are developing for logic programming and have turned out to be of practical use for software development in PROLOG style.

Their purpose is not to be fully automatizable as in [KS 86, Fri 88] but to give the user some better ability to deal with and understand logic programs.

Partial correctness does not say anything about the existence of answer substitutions (completeness) or effective form of the goals during or after computations ("run-time properties" in [DrM 87] STP in [DF 88]), or whether any solution can be reached using some computation rule (termination, [FGK 85]) ...etc... All together these problems are part of the total correctness of a definite clause program. They have been considered in other works [also in DM 89] but for their solution many problems use partial correctness properties (or valid specifications), and, as such, the partial correctness proof methods are the basic elements of any general validation system for logic programming.

Sometimes it is argued that logic programs are axioms, hence they do not need to be verified. However, this is not the case. In particular programmers frequently write general axioms which are used in a restricted manner. This is mainly due to the high level of expression permitted by the language.

Consider for example the program "concatenate" using difference-lists given by the unique clause :

concatenate (L1- L2, L2-L3, L1-L3) $\leftarrow$.

In its logical semantics [AvE 82, Llo 84] (we use in place the denotation, i.e the set of all atomic logical consequences [Cla 79, Fer 85]) there are not only difference-lists : for example concatenate([ ]-[1], [1]-[1, 2], [ ]-[1, 2]) is a logical consequence also. But it is sufficient (for partial correctness purposes only) from the programmers point of view to know that concatenate is correct w.r.t. the fact that if the two first arguments are difference-lists representing lists l1 and l2, then (if the goal succeeds) the third argument is a difference-list representing the concatenation of l1 and l2.

Proof of partial correctness are thus required to certify the validity of the written axioms in the specific model the user has in mind. In practice, however, one does not expect to write axioms for any kind of model but more frequently axioms of a specific model.

The purpose of this paper is to rephrase the proof methods developed in [CD 88] for Attribute Grammars and Logic Programming without unnecessary references to Attribute Grammars.Two proof methods are presented. Both are proven, sound and complete (provided the specification language is large enough [Coo 78]). The first one consists of defining a specification stronger than the original one, which furthermore is inductive (fix-point induction).

Our second method is a refinement of the first one : with every predicate we associate a finite set of formulas (we call this an annotation), together with implications between formulas. The proofs become more modular and tractable, but its consistency has to be proven, this is a decidable property. This method is particularly suitable for proving the validity of specifications which are not inductive.

The proof methods presented here are a direct adaptation of the proof methods described in [Der 83, Cou 84, CD 88] for attribute grammars to the case of logic programming. In [CD 88] the adaptation is sketched using the results of [DM 85] : a definite clause program can be thought of as an attribute grammar. In this paper the methods are defined directly in the logic programming framework without unnecessary references to the attribute grammar field, and the formulation of the second method is more general than in [CD 88] such that many applications concerning the proof of other properties like "run time" or safe use of the negation can thus be considered.

The paper is organized as follows :

Section 1 recalls basic definitions of logical languages used to describe the specifications, section 2 defines the notion of validity of a specification for a definite clause program. Section 3 and 4 present and study the two proof methods.

It is important to remark that even if the semantics of logic programs is usually defined on non sorted algebras of terms, the specifications may be expressed in any logical language in which terms are interpreted. For that reason we use many sorted languages to describe the specifications of the programs. Obviously presentation using homogeneous algebras could be also made.

1 - Basic definitions and notations

(1.1) Sort, signatures, terms

Context free grammars and logical languages will be defined in the algebraic style of [CD 88]. Some definitions are also useful to describe logic programs.

Let S be a finite set of sorts. A S-sorted signature F is a finite set of function symbols with two mappings : the arity α (some word in S* representing the sorts of the arguments in the same order), the sort σ of the function symbol. The length of $\alpha(f)$ is called the rank of f and denoted $\rho(f)$. If $\alpha(f) = \varepsilon$ (the empty word) then f is a constant symbol. The pair $< \alpha(f), \sigma(f) >$ is the profile of f. A constant of sort s has profile $< \varepsilon, s >$.

A heterogeneous F-algebra is an object $\mathbb{A}$:

$$\mathbb{A} = < \{A_s\}_{s \in S}, \{f_A\}_{f \in F} >$$

where $\{A_s\}$ is a family of non empty sets indexed by S (the carriers) and each f_A a mapping :

$$A_{s1} \times \ldots \times A_{sn} \rightarrow A_s \text{ if f has profile } < s1...sn, s >.$$

Let V be a S-sorted set of variables (each v in V has arity ε, sort $\sigma(v)$ in S). The free F-algebra $\mathbb{T}$ generated by V, also denoted T(F, V) is identified as usual as the set of the well-formed terms, "well typed" with respect to sorts and arities. Terms will also be identified with trees in a well known manner. T(F) denotes the set of all terms without variables, i.e. the ground terms, $T(F)_s$ denotes the set of the ground terms of sort s.

A term t in $T(F)_s$ is considered as denoting a value t_A in A_s for a F-algebra $\mathbb{A}$. For any F-algebra $\mathbb{A}$ and a S-sorted set of variables, an assignment of values in A_s to variables V_s, for all s in S, is an S-indexed family of functions.

$$\nu = \{ \nu_s : V_s \rightarrow A_s \}_{s \in S}$$

It is well-known that this assignment can be extended into a unique homomorphism

$$\nu' = \{ \nu'_s : T(F, V)_s \rightarrow A_s \}_{s \in S}$$

In $\mathbb{T}$ assignments are called substitutions. For any assignment ν in $\mathbb{T}$ and term t in T(F, V), νt is called an instance of t.

(1.2) Grammars

Proof-trees of a logic program can be thought of as abstract syntax trees with associated atoms. These abstract syntax trees can be represented by abstract context free grammars. An abstract context free grammar is the pair < N, P > where N is a finite set (the non-terminal alphabet) and P a N-sorted signature (for more details see [DM 85]).

(1.3) Many sorted logical languages

The specification will be given in some logical language together with an interpretation that we define as follows. Let S be a finite set of sorts containing the sort bool of the boolean values true, false. Let V be a sorted set of variables, F a S-signature and R a finite set of many sorted relation symbols (i.e. a set of symbols, each of them having an arity and, implicitely, the sort bool).

A logical language L over V, F, R is the set of formulas written with V, F, R and logical connectives like $\forall, \exists, \Rightarrow, \wedge, \vee$, not, ... We denote by free (φ) the possibly empty set of the free variables of the formula φ of L (free (φ) $\subseteq$ V), by AND A (resp OR A) the conjunction (resp. the disjunction) of formulas (AND $\emptyset$ = true, OR $\emptyset$ = false), and by $\varphi\ [t_1/v_1, \ldots, t_n/v_n]$ the result of the substitution of t_i for each free occurrence of v_i (some renaming of variables may be necessary). We do not restrict a priori the logical language to be first order.

Let $\mathcal{C}$(L) denote a class of L-structures, i.e. objects of the form :

$$\mathbb{D} = < \{D_s\}_{s \in S}\ , \{f_D\}_{f \in F}, \{r_D\}_{r \in R} >$$

where $< \{D_s\}_{s \in S}\ , \{f_D\}_{f \in F} >$ is a heterogeneous F-algebra and for each r in R, r_D is a total mapping

$$D_{s1} \times \ldots \times D_{sn} \rightarrow \{\underline{true}, \underline{false}\ \} = \underline{bool} \quad \text{if } \alpha(r) = s1 \ldots sn.$$

The notion of validity is defined in the usual way. For every assignment ν, every $\mathbb{D}$ in $\mathcal{C}$(L), every φ in L, one assumes that $(\mathbb{D}, \nu) \models \varphi$ either holds or does not hold. We say φ is valid in $\mathbb{D}$, and write $\mathbb{D} \models \varphi$, iff $(\mathbb{D}, \nu) \models \varphi$ for every assignment ν.

2 - Definite Clauses Programs, specifications

(2.1) Definition : Definite Clause Program (DCP).

A DCP is a triple P = <PRED, FUNC, CLAUS> where PRED is a finite set of predicate symbols, FUNC a finite set of function symbols disjoint of PRED, CLAUS a finite set of clauses defined as usual [Cla 79, Llo 87] with PRED and TERM = T(FUNC, V). Complete syntax can be seen in examples (2.2) and (2.3). A clause is called a fact if it is restricted to an atomic formula. An atomic formula is built as usual with PRED and TERM.

(2.2) Example

PRED	=	{plus}	ρ(plus) = 3
FUNC	=	{zero, s}	ρ(zero) = 0, ρ(s) = 1
CLAUS	=	{c1 : plus (zero, X, X) $\leftarrow$,	
		c2 : plus (s(X), Y, s(Z)) $\leftarrow$ plus (X, Y, Z)}	

variables begin with uppercase letters.

(2.3) Example "List" terms are represented in Edinburgh syntax

PRED = {perm, extract} $\rho(\text{perm}) = 2$, $\rho(\text{extract}) = 3$

FUNC = {[],[_|_]} $\rho([]) = 0$, $\rho([\_|\_]) = 2$

CLAUS = {c1 : perm ([], []) $\leftarrow$,
c2 : perm ([A|L], [B|M]) $\leftarrow$ perm(N, M), extract([A|L], B, N) ,
c3 : extract ([A|L], A, L) ,
c4 : extract ([A|L], B, [A|M]) $\leftarrow$ extract (L, B, M)}

(2.4) Definition : denotation of a DCP P : DEN(P)

The denotation of a DCP is the set of all its atomic logical consequences :

$$\text{DEN(P)} = \{a \mid P \vdash a\}$$

We do not give any more details on the notions of models of P (structures in which the clauses are valid formulas) and of logical consequences (all atoms of DEN(P) are valid in the models of P), since we won't make use in this paper of the logical semantics of a logic program, but rather of its constructive semantics that we shall now define. Other details can be found in [Cla 79, Ave 82, Llo 87, Fer 85].

(2.5) Definition : proof-tree [Cla 79, DM 85]

A proof-tree is an ordered labeled tree whose labels are atomic formulae (possibly including variables). The set of the proof-trees of a given DCP P = < PRED, FUNC, CLAUS > is defined as follows :

1 - If A $\leftarrow$ is an instance of a fact of CLAUS (instances built with TERM), then the tree consisting of one vertice with label A is a proof-tree.

2 - If $T_1,\ldots, T_q$ for some $q > 0$ are proof-trees with roots labeled $B_1,\ldots, B_q$ and if $A \leftarrow B_1,\ldots, B_q$. is an instance of a clause of CLAUS, then the tree consisting of the root labeled with A and the subtrees $T_1,\ldots, T_q$ is a proof-tree.

By a partial proof-tree we mean any finite tree constructed by "pasting together" instances of clauses. Thus a proof-tree is a partial proof-tree all of whose leaves are instances of facts. We denote by PTR(P) the set of all root labels of proof-trees of P, in short the proof-tree roots of P. Note that every instance of a proof-tree is a proof-tree.

(2.6) Proposition [Cla 79, Fer 85, DF 86a] - Constructive semantics
Given a DCP P : DEN(P) = PTR(P).

Thus, instead of the logical semantics of a logic program, one can deal with its constructive semantics. As pointed out in [DM 85, DM 88], proof-trees can be thought as syntax trees (terms of a clauses-algebra) "decorated" by atoms as specified in the proof-tree definition. Thus inductive proof methods as defined in [CD 88] may be applied to logic programs. This will be done in the next section. All the definitions are adapted from [CD 88] to the logic programming case.

(2.7) <u>Definition</u> : <u>Specification</u> $\mathbf{S}$ on $(L, \mathbb{D})$ of a logic program P.

A specification of a logic program P is a family of formulas $\mathbf{S} = \{ \mathbf{S}^p \}_{p \in \mathrm{PRED}}$ of a logical language L over V, F, R such that V contains the variables used in P and F contains FUNC, together with a L-structure $\mathbb{D}$. For every p of PRED, we denote by <u>varg</u>(p) = { p1, ..., pρ(p) } the set of variable names denoting any possible term in place of the 1st, ... or $\rho(p)$th argument of p. Thus we impose <u>free</u>($\mathbf{S}^p$) $\subseteq$ <u>varg</u>(p). Variables in a specification also begins with an uppercase letter.

(2.8) <u>Definition</u> : <u>valid specification</u> $\mathbf{S}$ for the DCP P.

A specification $\mathbf{S}$ on $(L, \mathbb{D})$ is <u>valid for</u> the DCP P (or P is <u>correct w.r.t</u> $\mathbf{S}$) iff

$$\forall p(t_1, ..., t_n) \in \mathrm{DEN}(P), \quad \mathbb{D} \models \mathbf{S}^p[\, t_1/p_1, ..., t_n/p_n], \ n = \rho(p).$$

In the following the notation will be abbreviated into $\mathbf{S}^p\,[t_1, ..., t_n]$ if no confusion may arise.

This means that every atom of the denotation satisfies the specification, hence every atom in any proof-tree. It also means that every answer substitution (if any) satisfies the specification. It corresponds to a notion of partial correctness referring to the declarative (i.e. constructive or logical) semantics since nothing is specified about the existence of proof-trees (the denotation can be empty), the way to obtain them or the kind of resulting answer substitution for a given goal.

(2.9) <u>Example</u> : specification for (2.2) :

L_1 = V_1 contains <u>varg</u>(plus) = {plus1, plus2, plus3}
F_1 = {zero, s, + }
R_1 = { = }

$\mathbb{D}_1$ = $\mathbb{N}$ the natural integers with usual meaning, zero is interpreted as 0, s as the increment by 1, + as the addition. (i.e. $\mathbb{N}$ is the L1-structure $\mathbb{D}$1)

$\mathbf{S}_1$ = $\{ \mathbf{S}_1^{plus} \}$, $\mathbf{S}_1^{plus}$: plus3 = plus1 + plus2

The validity of $\mathbf{S}_1$ (which is proved in the next section) means that the program "plus" in (2.2) specifies the addition on $\mathbb{N}$, or that every n-uple of values corresponding to the interpreted arguments of the elements of the denotation satisfy the specification plus3 = plus1+ plus2 .

L_2 = V_2 as in L_1 , F_2 = { zero, s}
R_2 = { ground } ρ(ground) = 1.

$\mathbb{D}_2$ contains the term algebra $T(F_2, V_2)$, and ground(t) is true iff t is a ground term.

$\mathbf{S}_2$ = $\{ \mathbf{S}_2^{plus} \}$, $\mathbf{S}_2^{plus}$: ground(plus3) $\Rightarrow$ [ground(plus1) $\wedge$ ground(plus2)]
$\mathbf{S}_2$ is a valid specification (it can be observed on every proof-tree and will be proven in the next section).

(2.10) Example : specification for (2.3). This example uses a many sorted L_3 structure.

L_3 = V_3 contains varg(perm) and varg(extract).
F_3 = { [], [_|_], nil, . , append } [] and nil are constants, the other operators have rank 2.
R3 = { is-a-list, permut } ρ(is-a-list) = 1, ρ(permut) = 2.

$\mathbb{D}_3$ contains two domains : list the usual domain of the lists of unspecified arguments, any the domain of the unspecified arguments.

the profiles are :

for functions
[] $\in$ < ε, list >, is interpreted as nil .
[_|_] $\in$ < any list, list > , is interpreted as . (cons of LISP)
nil $\in$ < ε, list > (as usual)
. $\in$ < any list, list > (as usually : cons of LISP)
append $\in$ < list list, list > (lists concatenation).

for relations
permut $\in$ <list list, bool > it is the relation defining the pairs of permuted lists.

S_3 = { S^{perm} : permut (perm1, perm2) ,
$S^{extract}$: $\exists$ L1, L2 (extract1 = append (L1, extract2.L2) $\wedge$ extract3 = append (L1, L2) }

3 - Inductive proof method

(3.1) Definition : Inductive specification S of a DCP P.

A specification S on (L, $\mathbb{D}$) of a DCP P is inductive iff for every c in CLAUS,

$$c: r_0(t01, ..., t0n_0) \Leftarrow r_1(t11, ..., t1n_1), ..., r_m(tm1, ..., tmn_m) ,$$
$$\mathbb{D} \models (\underset{1 \leq k \leq m}{\text{AND}} S^{rk} [tk1, ..., tkn_k] \Rightarrow S^{r0} [t01, ..., t0n_0])$$

i.e. a specification is inductive iff in every clause, if the specification holds for the atoms of the body, it holds for the head.

(3.2) Proposition :

If S of P is inductive then S is valid for P .

Proof : by an easy induction on the size of the proof-trees, if $PTR_n(P)$ denotes the set of all roots of the proof-tree of size $\leq$ n, S holds in $PTR_{n+1}(P)$ (by definition (2.5) and (3.1) and the notion of validity), thus in DEN(P) = $\bigcup_{n \geq 0} PTR_n(P)$. QED.

(3.3) Definition : strongest (weakest) specification of P

Let S and S' be two specifications of P on (L, $\mathbb{D}$). One says that S is weaker than S' (or S' stronger than S), and denotes it by $\mathbb{D} \models (S' \Rightarrow S)$ iff $\forall p \in$ PRED, $\mathbb{D} \models (S'^p \Rightarrow S^p)$.

Given a program P, we consider an L-structure $\mathbb{D}$ such that $\mathbb{D}$ contains an interpretation of FUNC. Then we consider L' the language of all relations on $\mathbb{D}$. Obviously $L \subseteq L'$. Note that L' may not be first order. We denote S_P the following specification on (L', $\mathbb{D}$) defined as :

$\forall p, t_1, ..., t_n \quad (\mathbb{D} \models S_P{}^p [t_1, ..., t_n])$ iff $p(t_1, ..., t_n) \in$ DEN(P).

We denote $S_{\underline{true}}$ the specification such that $S^p_{\underline{true}}$: $\underline{true}$ for all p in PRED (i.e no specification).

(3.4) <u>Proposition</u> : Given a DCP P, S_P and $S_{\underline{true}}$ are respectively the <u>strongest</u> and the <u>weakest</u> valid specification for P and S_P is inductive i.e. all valid specifications S satisfy :

$\mathbb{D} \models (S_P \Rightarrow S \Rightarrow S_{\underline{true}})$, and S_P is inductive.

<u>Proof</u> : it is easy to observe that S_P on (L', $\mathbb{D}$) is inductive, hence valid and that every valid specification S on (L, $\mathbb{D}$) – thus on (L', $\mathbb{D}$) – satisfies $\mathbb{D} \models S_P \Rightarrow S$. Obviously $\mathbb{D} \models S \Rightarrow S_{\underline{true}}$.

(3.5) <u>Theorem</u> (soundness and completeness of the inductive proof method)

A specification S on (L, $\mathbb{D}$) is valid for P <u>iff</u> it is weaker than some inductive specification S' on (L', $\mathbb{D}$) :

i.e. 1) $\exists$ S' inductive

2) $\mathbb{D} \models S' \Rightarrow S$.

<u>Proof</u> : Soundness is trivial since if S' is inductive, it is valid by proposition (3.2) and if $\mathbb{D} \models S' \Rightarrow S$ and S' true, then S is also true.

Completeness is stated in proposition (3.4) with $S' = S_P$. Note that one does not have the completeness if one restricts the specification language like first order logic (Proof of this claim will be given in an extended version of this paper).

(3.6) <u>Example</u> (2.2): the specification S_1 (2.9) is inductive.

Following the definition (3.1) it is sufficient to prove :

$$\mathbb{N} \models S_1[\text{ zero}, X, X] \text{ and}$$

$$\mathbb{N} \models S_1[X, Y, Z] \Rightarrow S_1[s(X), Y, s(Y)]$$

thus:

$$\mathbb{N} \models 0 + X = X \text{ and}$$

$$\mathbb{N} \models (X + Y = Z \Rightarrow X+1 + Y = Z+1)$$

which are valid formulas on $\mathbb{N}$.

(3.7) Example (2.2): the specification S_2 (2.9) is inductive.

In the same way it is easy to show that the following formulas are valid on $\mathbb{D}_2$:

$$\text{ground}(0) \wedge \text{ground}(X) \Rightarrow \text{ground}(X)$$

$$[\text{ground}(Z) \Rightarrow (\text{ground}(X) \wedge \text{ground}(Y))] \Rightarrow [\text{ground}(s(Z)) \Rightarrow (\text{ground}(s(X)) \wedge \text{ground}(Y)]$$

(3.8) Example (2.3): the specification S_3 (2.10) is inductive.

It is easy to show that the following formulas are valid on $\mathbb{D}_3$: (some replacements are already made in the formulas and universal quantifications on the variables are implicit)

in c1 : permut (nil, nil)

in c2 : permut (N, M) $\wedge$ $\exists$ L1, L2 (A.L = append(L1, B.L2) $\wedge$ N = append (L1, L2))
$\Rightarrow$ permut (A.L, B.M)

in c3 : $\exists$ L1, L2 (A.L = append(L1, A.L2) $\wedge$ L = append (L1, L2))
(L1 and L2 are lists) take L1 = nil and L2 = L.

in c4 : $\exists$ L1, L2 L = append(L1, B.L2) $\wedge$ M = append (L1, L2))
$\Rightarrow$ $\exists$ L'1, L'2 (A.L = append(L'1, B.L'2) $\wedge$ A.M = append (L'1, L'2))

take L'1 = A.L1 and L'2 = L2.

(3.9) Example:

We achieve this illustration by a proof of the claim in the introduction concerning the program concatenate :

$S_{concatenate}$ = repr(concatenate3) = append(repr(concatenate1), repr(concatenate2))

defined on (L_3, $\mathbb{D}_3$) enriched with two operators: "-" of profile <list list, diflist> and "repr" of profile <diflist, list>, defining the list represented by a difference list.

The claim is obvious as
repr(L1-L3) = append(repr(L1- L2), repr(L2 -L3))

(3.10) Remark that as noticed in [CD88] this proof method can be viewed as a fix point induction on logic programs. It seems very easy to use, at least on simple programs, which are sometimes very hard to understand. The ability of the programmer to use this method may improve his ability to understand and thus handle axioms of logic programs.

4 - Proof method with annotations

The practical usability of the proof method of theorem (3.5) suffers from its theoretical simplicity : the inductive specifications $\mathbf{S}'$ to be found to prove the validity of some given specification $\mathbf{S}$ will need complex formulas $\mathbf{S}'^{p}$ since we associate only one for each p in PRED. It is also shown in [CD 88] that $\mathbf{S}'$ may be exponential in the size of the DCP (to show this result we can use the DCP's transformation into attribute grammars as in [DM 85]). The proof method with annotations is introduced in order to reduce the complexity of the proofs : the manipulated formulas are shorter, but the user has to provide the organization of the proof i.e. how the annotations are deducible from the others. These indications are local to the clauses and described by the so called logical dependency scheme. It remains to certify the consistency of the proof, i.e. that a conclusion is never used to prove itself. Fortunately this last property is decidable and can be verified automatically, using the Knuth algorithm [Knu 68] or its improvements [DJL 86].

(4.1) Definition : annotations of a DCP

Given a DCP P, an annotation is a mapping Δ assigning to every p in PRED a finite set of formulas or assertions $\Delta(p)$ built as in definition (2.7). It will be assumed that assertions are defined on $(L, \mathbb{D})$.

The set $\Delta(p)$ is partitioned into two subsets $I\Delta(p)$ (the set of the inherited assertions of p) and $S\Delta(p)$ (the set of the synthesized assertions of p).

The specification $\mathbf{S}_\Delta$ associated with Δ is the family of formulas :

$$\{ \mathbf{S}^{p}_{\Delta} : \mathbb{AND}\ I\Delta(p) \Rightarrow \mathbb{AND}\ S\Delta(p) \}_{p \in \text{PRED}}$$

(4.2) Definition : validity of an annotation Δ for a DCP P.

An annotation Δ is valid for a DCP P iff for all p in PRED in every proof-tree T of root $p(t_1, ..., t_{np})$: if $\mathbb{D} \models \mathbb{AND}\ I\Delta(p)\ [t_1, ..., t_{np}]$ $(np = \rho(p))$ then every label $q(u_1, ..., u_{nq})$ $(nq = \rho(q))$ in the proof-tree T satisfies : $\mathbb{D} \models \mathbb{AND}\ \Delta(q)\ [u_1, ..., u_{nq}]$.

In other words, an annotation is valid for P if in every proof-tree whose root satisfies the inherited assertions, all the assertions are valid at every node in the proof-tree, hence the synthesized assertions of the root.

(4.3) Proposition : if an annotation Δ for the DCP P is valid for P, then $\mathbf{S}_\Delta$ is valid for P.

Proof : It follows from the definition of $\mathbf{S}_\Delta$, the definitions of validity of an annotation (4.2) and of a specification (2.8).

Note that $\mathbf{S}_\Delta$ can be valid but not inductive (see example (4.14)).

We shall give sufficient conditions insuring the validity of an annotation and reformulate the proof method with annotations. This formulation is slightly different from that given in [CD 88]. The introduction of the proof-tree grammar is a way of providing a syntaxic formulation of the organization of the proof.

(4.4) <u>Definition</u> : <u>Proof-tree grammar</u> (PG)

Given a DCP P = < PRED, FUNC, CLAUS >, we denote Gp and call it the <u>proof-tree grammar of P</u>, the abstract context free grammar < PRED, RULE > such that RULE is in bijection with CLAUS and r of RULE has profile $< r_1 r_2 \ldots r_m, r_0 >$ iff the corresponding clause in CLAUS is $c : r_0(\ldots) \leftarrow r_1(\ldots), \supseteq \ldots, r_m(\ldots)$.

Clearly a (syntax) tree in Gp can be associated to every proof-tree of P. But not every tree in Gp is associated a proof-tree of P.

(4.5) <u>Definition</u> : <u>Logical dependency scheme for Δ</u> (LDS_Δ).

Given a DCP P and an annotation Δ for P, a logical dependency scheme for Δ is $LDS_\Delta = < Gp, \Delta, D >$ where Gp = < PRED, RULE > is the proof-tree grammar of P and D a family of binary relations defined as follows.

We denote for every rule r in RULE of profile $< r_1 r_2 \ldots r_m, r_0 >$
$W_{\underline{hyp}}(r)$ (resp. $W_{\underline{conc}}(r)$) the sets of the hypothetic (resp. conclusive) assertions which are :
$W_{\underline{hyp}}(r) = \{\varphi_k \mid k = 0, \varphi \in I\Delta(r_0) \text{ or } k > 0, \varphi \in S\Delta(r_k)\}$
$W_{\underline{conc}}(r) = \{\varphi_k \mid k = 0, \varphi \in S\Delta(r_0) \text{ or } k > 0, \varphi \in I\Delta(r_k)\}$

where φ_k is φ in which the free variables <u>free</u> (φ) = $\{p_1, \ldots, p_n\}$ have been renamed into <u>free</u> (φ_k) = $\{p_{k1}, \ldots, p_{kn}\}$.

The renaming of the free variables is necessary to take into account the different instances of the same predicate (if $r_i = r_j$ = pr in a clause for some different i and j) and thus different instances of the same formula associated with pr, but this will not be explicit anymore by using practically the method.

$D = \{D(r)\}_{r \in RULE}, D(r) \subseteq W_{\underline{hyp}}(r) \times W_{\underline{conc}}(r)$.

From now on we will use the same name for the relations D(r) and their graph. For a complete formal treatment of the distinction see for example [CD 88]. We denote by <u>hyp</u> (φ) the set of all formulas ψ such that (ψ, φ) $\in$ D(r) and by <u>assoc</u> (φ) = p($t_1, \ldots, t_n$) the atom to which the formula is associated by Δ in the clause c corresponding to r.

(4.6) <u>Example</u> : annotation for example (2.2) and specification S_2 (2.9).

Δ(plus)	=	$I\Delta$(plus) $\cup$ $S\Delta$(plus)
$I\Delta$(plus)	=	{φ : ground (Plus3)}
$S\Delta$(plus)	=	{ψ : ground (Plus1), δ : ground (Plus2)}
$S_\Delta$plus	=	S_2plus

G_{plus} : PRED = {plus}
RULE = $\{r_1 \in <\varepsilon, plus>, r_2 \in <plus, plus>\}$

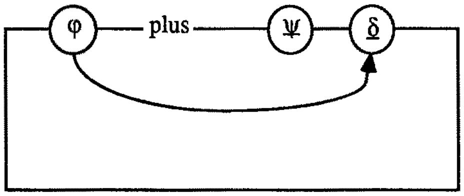
D : $D(r_1) = \{\varphi_0 \rightarrow \delta_0\}$
$D(r_2) = \{\varphi_0 \rightarrow \varphi_1, \psi_1 \rightarrow \psi_0, \delta_1 \rightarrow \delta_0\}$ (see the scheme below)

$W_{\underline{hyp}}(r_1) = \{\varphi_0\}, W_{\underline{conc}}(r_1) = \{\psi_0, \delta_0\}$
$W_{\underline{hyp}}(r_2) = \{\varphi_0, \psi_1, \delta_1\}, W_{\underline{conc}}(r_2) = \{\varphi_1, \psi_0, \delta_0\}$

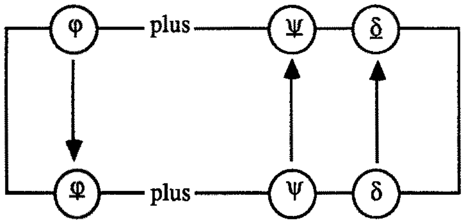
Note that in r_2 for example :
φ_0 = ground (Plus03)
φ_1 = ground (Plus13)...

In order to simplify the presentation of D we will use schemes as in [CD 88] representing the rules in RULE and the LDS of Δ. Elements of $W_{\underline{conc}}$, will be underlined. Inherited (synthesized) assertions are written on the left (right) hand side of the predicate name. Indices are implicit : 0 for the root, 1... to n following the left to right order for the sons.

r 1 :



r 2 :



(4.7) <u>Definition</u> : <u>Purely-synthesized LDS, well-formed LDS</u>.

A LDS for Δ is <u>purely-synthesized</u> iff IΔ = Ø, i.e. there are no inherited assertions.

A LDS for Δ for P is <u>well-formed</u> iff in every tree t of Gp the relation of the induced dependencies D(t) is a partial order (i.e. there is no cycle in its graph).

To understand the idea of well-formedness of the LDS it is sufficient to understand that the relations D(r) describe dependencies between instances of formulas inside the rules r. Every tree t of Gp is built with instances of rules r in RULE, in which the local dependency relation D(r) defines dependencies between instances of the formulas attached to the instances of the non-terminals in the rule r. Thus the dependencies in the whole tree t define a new dependency relation D(t) between instances of formulas in the tree. A complete treatment of this question can be found in [CD 88]. We recall here only some important results [see Knu 68, DJL 88 for a survey on this question] :

(4.8) Proposition - The well-formedness property of an LDS in decidable.
- The well-formedness test is intrinsically exponential.
- Some non trivial subclasses of LDS can be decided in polynomial time.
- A purely-synthesized LDS is (trivially) well-formed.

(4.9) Definition : Soundness of a LDS for Δ.

A LDS for Δ < G_p, Δ, D > is sound iff for every r in RULE and every
φ in $W_{conc}(c)$ with assoc $(\varphi) = q(u_1, ..., u_{nq})$ the following holds :
$\mathbb{D} \models \text{AND}\{ \psi[t_1, ..., t_{np}] \mid \psi \in \text{hyp}(\varphi) \text{ and } \text{assoc}(\psi) = p(t_1, ..., t_{np}) \} \Rightarrow \varphi[u_1, ..., u_{nq}]$
(Note that the variable q_i (p_i) in a formula φ (ψ) is replaced by the corresponding term u_i (t_i)).

(4.10) Example : The LDS given in example (4.6) is sound. In fact it is easy to verify that the following holds in $\mathbb{T}$:

in r1 :	ground(X)	⇒ ground(X)
	ground(0)	
in r2:	ground(sX)	⇒ ground(X)
	ground(Y)	⇒ ground(Y)
	ground(Z)	⇒ ground(sZ)

(4.11) Theorem : Δ is valid for P if it exists a sound and well-formed LDS_Δ for Δ for P.

Sketch of the proof by induction on the relation D(t) induced in every proof-tree, following the scheme given in [Der 83] or the proof given in [CD 88, theorem (4.4.1)]. The only difference comes from the lack of attribute definitions replaced by terms. Notice that the free variables appearing in the formulas (4.9) are the free variables of the corresponding clause c. They are also quantified universally. Hence the results, as a proof-tree is built with clause instances. In fact, the implications will hold also in every instance of a clause in the proof-tree as the variables appearing in a proof-tree can be viewed universally quantified (every instance of a proof-tree is a proof-tree). QED.

(4.12) Theorem (soundness and completeness of the annotation method for proving the validity of specifications) : We use the notations of (3.3) and (3.5).

A specification **S** on (L, $\mathbb{D}$) is valid iff it is weaker than the specification $\mathbf{S}_\Delta$ of an annotation Δ on (L', $\mathbb{D}$) with a sound and well-formed LDS (L' as in 3.3).

i.e. :

1) there exists a sound and well-formed LDS_Δ.
2) $\mathbb{D} \models \mathbf{S}_\Delta \Rightarrow \mathbf{S}$.

Proof (soundness) follows from theorem (4.11).
(completeness) follows from the fact that $\mathbf{S}_P$ on (L', $\mathbb{D}$) is a purely synthesized (thus well-formed) sound annotation.

We complete this presentation by giving some examples.

(4.13) Example (4.10) continued.

The LDS is sound and well-formed, thus $S_\Delta{}^{plus} = S_2{}^{plus}$ is a valid specification.

(4.14) Example We borrow from [KH 81] an example given here in a logic programming style : it computes multiples of 4.

c1 : fourmultiple (K) $\leftarrow$ p(0, H, H, K).

c2 : p(F, F, H, H) $\leftarrow$

c3 : p(F, sG, H, sK) $\leftarrow$ p(sF, G, sH, K)

$S^{fourmultiple}$: $\exists$ N, N $\geq$ 0 $\wedge$ Fourmultiple1 = 4$*$N

L, $\mathbb{D} = \mathbb{D}_1$ as in (2.9) enriched with $*$, ≥ 0 etc...

The following annotation Δ is considered in [KH 81] :

$I\Delta$ (fourmultiple) = $\emptyset$, $S\Delta$ (fourmultiple) = { $S^{fourmultiple}$} = {δ}

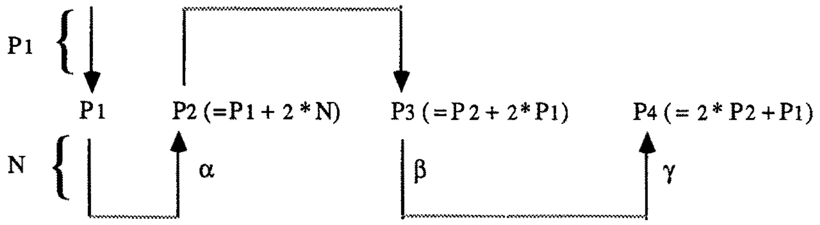
$I\Delta(p) = \{\beta\}$ $S\Delta(p) = \{\alpha, \gamma\}$

$\alpha : \exists N, N \geq 0 \wedge P2 = P1 + 2 * N$

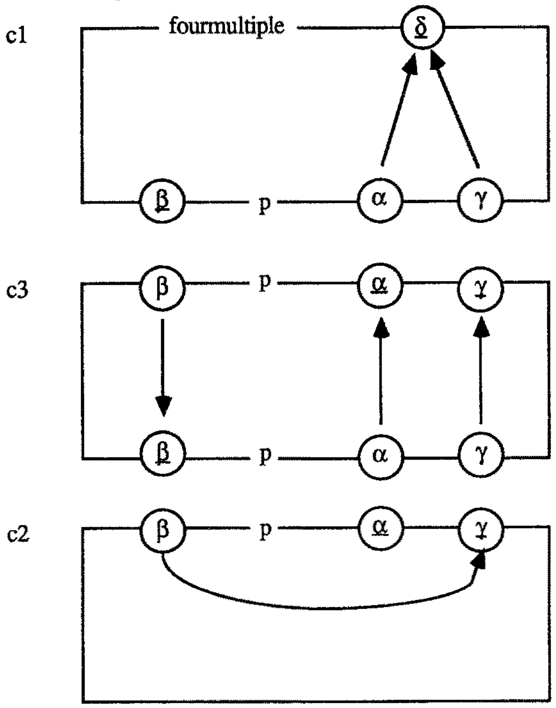
$\beta : P3 = P2 + 2 * P1$

$\gamma : P4 = 2 * P2 + P1$

The assertions can be easily understood if we observe that such a program describes the construction of a "path" of length 4$*$N and that p1, p2, p3 and p4 are lengths at different steps of the path as shown in the following figure :



The LDS for Δ is the following :



The LDS is sound and well-formed. For example it is easy to verify that the following fact holds in $\mathbb{D}_1$:

in c1 :

$(\alpha_1 \wedge \gamma_1 \Rightarrow S_0^{\text{fourmultiple}})$ that is : $\exists N, N \geq 0 \wedge H = 0 + 2*N \wedge K = 2*H + 0$

$\Rightarrow \exists N, N \geq 0 \wedge K = 4*N$

(β_1) that is : $H = H + 2*0$

in c2 :

$(\beta_0 \Rightarrow \gamma_0)$ that is : $H = F + 2*F \Rightarrow H = 2*F + F$

(α_0) that is : $\exists N, N \geq 0 \wedge F = F + 2*N$ (with $N = 0$)

etc...

Note that as S_Δ is inductive, this kind of proof modularization can be viewed as a way to simplify the presentation of the proof of S_Δ.

Now we consider on the same program a non inductive valid specification ξ defined on L_2, $\mathbb{D}_2$ (2.9) :

$\xi^{\text{fourmultiple}}$	: ground (Fourmultiple1)
ξ^{p}	: [ground(P1) $\wedge$ ground(P3)] $\Rightarrow$ [ground(P2) $\wedge$ ground(P4)]

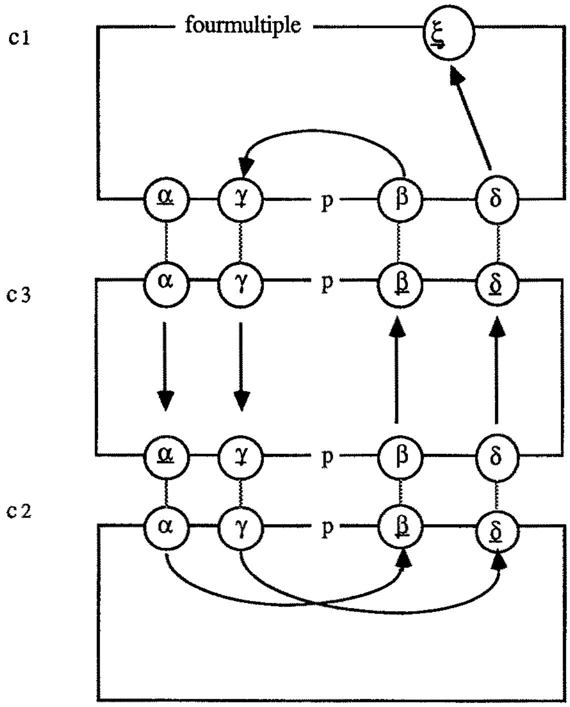
The specification clearly is valid but not inductive since the following does not hold with $\mathbb{D}_2$ (term-algebra) in c1

$\mathbb{D}_2 \not\models \xi p_1 \Rightarrow \quad \xi^{\text{fourmultiple}}{}_0$

i.e.

$\mathbb{D}_2 \not\models [\ (\text{ground}(0) \wedge \text{ground}(H)) \Rightarrow (\text{ground}(H) \wedge \text{ground}(K))\ ] \Rightarrow \text{ground}(K)$

But it is easy to show that the following LDS is sound and well-formed :



$I\Delta$ (fourmultiple) = $\emptyset$, $S\Delta$ (fourmultiple) = { $\xi^{\text{fourmultiple}}$}
$I\Delta$ (p) = $\{\alpha, \gamma\}$, $S\Delta$ (p) = $\{\beta, \delta\}$
α : ground(P1)
β : ground(P2)

γ : ground(P3)
δ : ground(P4)

Dotted lines help to observe that the LDS is well-formed (without circularities). The proofs are trivial.

Note that the corresponding inductive specification is $(\alpha \Rightarrow \beta) \wedge (\gamma \Rightarrow \delta)$. It is shown in [CD 88] how the inductive specification can be inferred from LDS_Δ.

Notice that this kind of proof of correctness corresponds to some kind of mode verification. It can be automatized for a class of programs identified in [DM 84] and experimentally studied in [Dra 87] (the class of simple logic programs). As shown in [DM 84] this leads to an algorithm of automatic (ground) modes computation for simple logic programs which can compute (ground) modes which are not inductive.

Conclusion

In this paper we have presented two methods for proving partial correctness of logic programs. Both are modular and independent of any computation rule.

These methods have two main advantages :

1) They are very general (complete) and simple (especially if a short inductive assertion is proven). As such they can be taught together with a PROLOG dialect and may help the user to detect useful properties of the written axioms. In the case of large programs the second method may help to simplify the presentation of a proof using shorter assertions and clear logical dependency schemes between assertions.

2) Valid specifications are the basic elements used in proofs of all other desirable logic program properties as completeness, "run-time" properties, termination such as shown in [DF 88] or safe use of the negation [Llo 87]. For example any proof of termination with regards to some kind of used goals and some strategy will suppose that, following the given strategy, some sub-proof-tree has been successfully constructed and thus that some previously chosen atoms in the body of a clause satisfy their specifications. Thus correctness proofs appear to be a way of making modular proofs of other properties also. In fact the validity of a specification can be established independently of any other property.

Work is at present in progress to adapt such methods to other properties of logic programs. These methods are currently being used to make proofs in the whole formal specification of standard PROLOG [DR 88].

Aknowledgments

We are indebted to B. COURCELLE with whom most of the basic ideas have been developed and to G. FERRAND and J.P. DELAHAYE who helped to clarify this text.

References

[AvE 82] K.R. Apt, M.H. Van Emden : Contributions to the theory of Logic Programming. JACM V29, N° 3, July 1982 pp 841-862.

[CD 88] B. Courcelle, P. Deransart : Proof of partial Correctness for Attribute Grammars with application to Recursive Procedure and Logic Programming. Information and Computation 78, 1, July 1988 (First publication INRIA RR 322 - July 1984).

[Cla 79] K.L. Clark : Predicate Logic as a Computational Formalism. Res. Mon. 79/59 TOC. Imperial College, December 1979.

[Coo 78] S.A. Cook : Soundness and Completeness of an Axiom System for Programs Verification. SIAM Journal. Comput. V7, n° 1, February 1978.

[Cou 84] B. Courcelle : Attribute Grammars : Definitions, Analysis of Dependencies, Proof Methods. In Methods and Tools for Compiler Construction, CEC-INRIA Course (B. Lorho ed.). Cambridge University Press 1984.

[Der 83] P. Deransart : Logical Attribute Grammars. Information Processing 83, pp 463-469, R.E.A. Mason ed. North Holland, 1983.

[Dev 87] Y. Deville : A Methodology for Logic Program Construction.PhD Thesis, Institut d'Informatique, Facultés Universitaires de Namur (Belgique), February 1987.

[DF 87] P. Deransart, G. Ferrand : Programmation en Logique avec Négation : Présentation Formelle. Publication du laboratoire d'Informatique, University of Orléans, RR 87-3 (June 1987).

[DF 88] P. Deransart, G. Ferrand : Logic Programming, Methodology and Teaching. K. Fuchi, L. Kott editors, French Japan Symposium, North Holland, pp 133-147, August 1988.

[DF 88] P. Deransart, G. Ferrand : On the Semantics of Logic Programming with Negation. RR 88-1, LIFO, University of Orléans, January 1988.

[DJL 88] P. Deransart, M. Jourdan, B. Lorho : Attribute Grammars : Definitions, Systems and Bibliography, LNCS 323, Springer Verlag, August 1988.

[DM 84] P. Deransart, J. Maluszynski : Modelling Data Dependencies in Logic Programs by Attribute Schemata. INRIA, RR 323, July 1984.

[DM 85] P. Deransart, J. Maluszynski : Relating Logic Programs and Attribute Grammars. J. of Logic Programming 1985, 2 pp 119-155. INRIA, RR 393, April 1985.

[DM 89] P. Deransart, J. Maluszynski : A Grammatical View of Logic Programming. PLILP'88, Orléans, France, May 16-18, 1988, LNCS 348, Springer Verlag, 1989.

[DR 88] P. Deransart, G. Richard : The Formal Specification of PROLOG standard. Draft 3, December 1987. (Draft 1 published as BSI note PS 198, April 1987, actually ISO-WG17 document, August 1988).

[Dra 87] W. Drabent, J. Maluszynski : Do Logic Programs Resemble Programs in Conventional Languages. SLP87, San Francisco, August 31 -September 4 1987.

[DrM 87] W. Drabent, J. Maluszynski : Inductive Assertion Method for Logic Programs. CFLP 87, Pisa, Italy, March 23-27 1987 (also : Proving Run-Time Properties of Logic Programs. University of Linköping. IDA R-86-23 Logprog, July 1986).

[Fer 85] G. Ferrand : Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Methods. INRIA, RR 375, March 1985. J. of Logic Programming Vol. 4, 1987, pp 177-198 (French version : University of Orléans, RR n° 1, August 1984).

[FGK 85] N. Francez, O. Grumberg, S. Katz, A. Pnuelli : Proving Termination of Prolog Programs. In "Logics of Programs, 1985", R. Parikh Ed., LNCS 193, pp 89-105, 1985.

[Fri 88] L. Fribourg : Equivalence-Preserving Transformations of Inductive Properties of Prolog Programs. ICLP'88, Seattle, August 1988.

[Hog 84] C.J. Hogger : Introduction to Logic Programming. APIC Studies in Data Processing n° 21, Academic Press, 1984.

[KH 81] T. Katayama, Y. Hoshino : Verification of Attribute Grammars. 8th ACM POPL Conference. Williamsburg, VA pp 177-186, January 1981.

[Knu 68] D.E. Knuth : Semantics of Context Free Languages. Mathematical Systems Theory 2, 2, pp 127-145, June 1968.

[KS 86] T. Kanamori, H. Seki : Verification of Prolog Programs using an Extension of Execution. In (Shapiro E., ed.), 3rd ICLP, LNCS 225, pp 475-489, Springer Verlag, 1986.

[Llo 87] J. W. Lloyd : Foundations of Logic Programming. Springer Verlag, Berlin, 1987.

[SS 86] L. Sterling, E. Y. Shapiro : The Art of Prolog. MIT Press, 1986.