Fundamental Study

Proof methods of declarative properties of definite programs

Pierre Deransart

INRIA, Domaine de Voluceau – Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

Communicated by M. Nivat
Received June 1990
Revised January 1992

Abstract


In this paper we shall consider proof methods for declarative properties of definite programs, i.e. properties holding at the proof-tree roots. It 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 program is partially correct iff all possible answer substitutions satisfy the specification.

This paper generalizes known results in logic programming in three ways: (i) it considers any kind of specification and any kind of domains of interpretation (in particular non-ground-extended Herbrand bases); (ii) its results can be applied to extensions of logic programming such as functions or constraints; (iii) a new proof method (the annotations method) is presented. It gives a unified framework for different kinds of known proof methods like the consequence verification method, the structural induction in definite clauses or the inductive proof of formulas.

Two proof methods are presented and proved to be sound and complete. The first is adapted from attribute grammar to logic programming. It defines a specification stronger than the original one, which, furthermore, is inductive. Many aspects of the inductive specification method are investigated. The second method is a refinement of the first one: with every predicate, one associates 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. It has the same power as the first method, but it is designed to prove properties holding inside the proof trees.

Correspondence to: P. Deransart, INRIA, Domaine de Voluceau – Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. Email: deransart@minos.inria.fr.
0. Introduction

The problem of proving the partial correctness of a definite program (in which the
clauses have exactly one positive literal) with respect to a given specification, saying
what the answer substitutions (if any) should be, has been considered many times
[12, 39, 30, 32, 49, 9] but rarely studied in depth. The purpose of this paper is to
present a unified view of the partial correctness proof methods of logic programs
based on the notion of proof of properties of the proof trees. It gives a unified
framework for different kinds of known proof methods like the consequence verifica-
tion method, the structural induction in definite clauses or the inductive proof of
formulas [12]. Also it introduces a new proof method to prove properties holding
inside the proof trees.

It is largely accepted that a definite program has an implicit actual semantics
(the set of its atomic logical consequences). For definite programs all the formaliz-
atations of the actual semantics coincide and we will consider that it is the set
of all the (not necessarily ground) proof trees. It is known that all the proof-tree
roots correspond to the atomic logical consequences of the program and conversely.
On the other hand, in writing a program, a programmer has some so-called intended semantics in mind. The intended semantics can also be formalized by a set
of atoms which must be compared with the actual semantics. Unfortunately, the
actual and the intended semantics do not coincide necessarily (if there is some bug). The practice of software development and the relatively low level of expres-
sion in logic programming require attention to validation methods. These are based
on the comparison of the actual semantics of a definite program and its intended semantics.

A program is partially correct w.r.t. its intended semantics if the relation it specifies satisfies also the intended semantics. To be as general as possible, by “intended semantics” we mean some property expressed in some logical language. Depending on
the degree of refinement of its expression, one may be interested in proving weaker or
stronger properties.
For example, consider the following definite program:

\[
\begin{align*}
&\text{plus}(\text{zero}, X, X), \\
&\text{plus}(s(X), Y, s(Z)) \leftarrow \text{plus}(X, Y, Z).
\end{align*}
\]

The property saying that “all the elements of the denotation have the form \(\text{plus}(s^n(\text{zero}), t, s^n(t))\) for \(n \geq 0, t\) being any term” is the strongest possible one. But one could be interested in proving only that “the first argument of plus has always the form \(s^n(\text{zero})\)” or “if the second argument is ground then the third is too and conversely”. These last propositions are a kind of partial specification or partial intended semantics. Another kind of “partial specification” which is considered in the literature \([12, 42]\) is to express some intended property of the program like the commutativity of “plus” with the given axioms interpreted on natural integers. This can be expressed by the formula

\[
\forall X, Y, Z \text{ integer}(X) \land \text{integer}(Y) \Rightarrow (\text{plus}(X, Y, Z) \Rightarrow \text{plus}(Y, X, Z)).
\]

We will show that all these properties can be proved using the same kind of induction on the structure of the program. We call this method the \textit{inductive assertion method}.

Our purpose is to do more. We introduce a new method to prove properties holding anywhere inside the proof trees: the \textit{annotation method}. This method is particularly well suited when restrictions are imposed at the root of the proof trees. It is thus a natural way to establish consequences inside the proof trees. However, it introduces also a simplification of the inductive assertion method by using modular assertions: In the inductive methods a big assertion is proved holding at the roots of the proof trees. In the annotation method shorter assertions will be used, instead of a big one, flowing through the proof tree. It is still a “syntax-directed” proof method but with “modular assertions”. It will be shown that the assertions used in an inductive proof method may have an exponential size w.r.t. the size of all the assertions used in an annotation method.

Furthermore, the annotation proof method is a solution for another kind of modularity: proof by stepwise refined assertions. Let us explain informally what it is. A property specifying (partially or completely) the intended semantics is expressed by one formula attached to each predicate and stating the relations holding between its arguments. Thus one assumes that the assertion language has an interpretation which includes all the functions and constant symbols of the program. Suppose now that instead of one formula, one attaches to each predicate two or more formulas. If one can prove independently that in all the proof trees all the “first” formulas hold, then one can prove the other ones assuming the first ones hold already. This is the idea of “stepwise refined assertions”. Let us illustrate it by an example.

Assume that one wants to prove that the program “plus” specifies an addition on natural integers. It seems natural to try to prove its correctness w.r.t. the specification associated with the relation

\[
\text{“plus}(X, Y, Z)\text{”}: Z =_{\text{int}} X +_{\text{int}} Y,
\]
i.e., the intended semantics is expressed as a formula

\[ Z = X + Y \]

in which the symbols "=" and "+" are, respectively, relational and functional symbols whose interpretation is defined for natural integers only but not for other values (there is no interpretation for \([ \ ] + \text{int} \ [ \ ],\) for example). In practice, partial functions and predicates are used.

If one assumes that the domains of interpretation are natural integers only, the proof by induction of such a formula is very easy to perform. This is the way such a program is usually understood when writing and reading its axioms. However, in performing such a proof, one makes assumptions about the type of the predicate arguments and some properties they satisfy (for example, we will see that one makes use of the commutativity of the operation "+" on natural integers). Such a proof assumes that if the program is well formed in the sense of the typing policy then all the proof trees can be interpreted on this (possibly many-sorted) interpretation; i.e., all the free variables in the clauses or the proof trees are quantified on the domains of their type, and the properties proved hold at the interpreted proof tree root.

This typing approach corresponds to some explicit hypothesis, which may be expressed as follows: \(\text{integer}(X) \land \text{integer}(Y) \land \text{integer}(Z) \Rightarrow Z = \text{int} X + \text{int} Y\), which is a property holding at all the proof tree roots (i.e., all the roots of the form \(\text{plus}(X, Y, Z)\)).

This kind of hypothesis can be viewed also as a stepwise refinement: first define a subset of proof trees which is well typed, then perform the rest of the proof assuming well-typedness. For this example it would be sufficient to assume that the second or third argument of \text{plus} is an integer. This illustrates a general practical situation in logic programming: a programmer is not interested in all the proof trees specified by his program but only in some of them.

For example the program \text{plus} given above, even if it is part of a larger program which uses other function symbols, will ordinarily be used assuming that all arguments are just integers. This hypothesis is not implicit by the given axioms of \text{plus}. In particular, in the context of lists, the atom \text{plus}(\text{zero}, [\ ], [\ ]) is a logical consequence of the axioms. However, under the hypothesis that "one of the second or the third argument of the root is an integer" any proof tree rooted by \text{plus} will be completely instantiated by integers. This can be formalized by an annotation, i.e., a set of formulas associated with the predicates. For example, with \text{plus}, three formulas of the form "the \{i}\{th} argument is an integer" and to which directions are assigned like: the second formula is inherited (hypotheses issued from the upper context), the others are synthesized (conclusions holding at the roots by assuming the hypotheses). Now we will say that a definite program is correct w.r.t. a given annotation (or the annotation is valid for the program) provided all the proof trees whose roots satisfy its hypothesis (inherited assertions) then all the assertions (inherited and synthesized) hold anywhere inside the proof tree (and, of course, at the root). The program \text{plus} is correct w.r.t. the
given annotation; that is, if the second argument of the root is an integer then the whole proof tree (if any) "contains" integers only.

Now it becomes clear – we hope – how the annotation method may be a factor of simplification: it is a way to organize the proof of a property in at least two independent steps: the verification of a well-typed subset of proof trees and the verification of some property assuming well-typedness.

These observations basically motivate the choices made in the presentation of this paper: a many-sorted approach is considered allowing one to handle assertions with partial functions in a very modular way. Here, we do not want just to obtain theoretical results but also to show how proofs can be organized such that they become simpler to handle and more tractable on big programs. However, note that it is not the purpose of this paper to formalize completely the many-sorted approach; When sorts will be used it will be done with the purpose of focussing on the most interesting parts of the assertions, assuming that the clauses are well formed and that a proof concerning the type of the arguments inside the proof trees of interest has been performed separately (this is illustrated by a detailed example in Section 5). Observe finally that this stepwise splitting of the correctness proofs into two parts is arbitrary. In general many steps could be performed.

We complete this introduction by a short presentation of the proof methods. Basically, this paper has two parts corresponding to the two proof methods: the proof by induction and the proof by annotations. Both are syntax-directed but, although the first can be viewed as a particular case of the second, it is studied separately. The reason is that most of the known literature is devoted to the first one and most of the basic theoretical results are obtained with it. The first method is devoted to the proof of partial correctness (validity of a specification) whereas the second is devoted to the proof of properties of the proof trees.

Let us state more formally the definition of partial correctness.

A specification consists of a logical formula associated with each predicate of a definite program $P$ and establishes a relation between its arguments. If $\mathcal{S}$ is the family of such logical formulas,

$$\mathcal{S} = \{ \mathcal{S}^p \mid \text{for each predicate } p \text{ in } P \},$$

and $\mathcal{D}$ the interpretation of the logical language including the functional symbols of $P$, then the partial correctness of $P$ with regard to $\mathcal{S}$ is expressed as follows:

$$\forall p \text{ predicate of } P, \forall \bar{t} \text{ list of term arguments, } \text{if } P \models p(\bar{t}) \text{ then } \mathcal{D} \models \mathcal{S}^p(\bar{t}),$$

which formally expresses that every atomic logical consequence of the program $P$ (the actual semantics) satisfies the specification $\mathcal{S}$.

The first method consists in defining a specification stronger than the original one, which, furthermore, is inductive. A specification is inductive if the axioms of the program are still valid in $\mathcal{D}$ when one replaces in the clauses the predicate occurrences by their specification. Many consequences and applications of the inductive view will
be explored, in particular those following from the induction principle contained in
the definite clauses of a logic program.

Our second method is a generalization of the first one: with every predicate we
associate a finite set of formulas (we call this an annotation), together with implica-
tions between formulas in the clauses. The proofs become more modular and tract-
able, but their consistency has to be proved; this is a decidable property. This method
is particularly suitable for proving the validity of specifications which are not induc-
tive, or to make type verifications inside the proof trees.

Let us consider the previous example with one more predicate whose intended
semantics is to specify the addition in natural integers and the length of a list:

\[
\text{plus}(\text{zero}, X, X) \leftarrow \\
\text{plus}(s(X), Y, s(Z)) \leftarrow \text{plus}(X, Y, Z).
\]

\[
\text{plength}([\ ], \text{zero}) \leftarrow \\
\text{plength}([A|L], N) \leftarrow \text{plength}(L, M), \text{plus}(\text{zero}, M, N).
\]

and the following specification: \( \varphi = \{ \varphi_{\text{plus}}, \varphi_{\text{plength}} \} \)

\[
\varphi_{\text{plus}}(X, Y, Z): Z = X + Y \\
\varphi_{\text{plength}}(L, N): N = \text{length}(L)
\]

assuming that \( X, Y, Z, N \) are integers and \( L \) a list.

We first prove the validity of the specification \( \varphi \) for the given program using the
first method, then we prove with the annotation method that the proof trees are well
typed.

This first specification is inductive (hence the program is correct w.r.t. \( \varphi \)). In fact
the axioms can be rewritten as

\[
X = 0 + X \\
Z = X + Y \Rightarrow Z + 1 = X + 1 + Y \\
\text{length}([\ ])=0 \\
\text{length}(L)=M \quad \text{and} \quad N = M + 1 \Rightarrow \text{length}(A|L])=N
\]

which are valid formulas in the domains of the natural integers \( (X, Y, Z, M, N) \), lists
\( (L) \) and unspecified elements \( (A) \).

Note that partial correctness does not say anything about the existence of solutions
(completeness) or effective form of the goals before, during or after computations
("run-time properties" in [33], "STP" in [22]), or whether any solution can be reached
using some computation rule (termination, [35, 3]). All together, these problems are
part of the total correctness of a definite clause program. Partial correctness states
only that any computed atom – if any – (which belongs necessarily to the actual
semantics) satisfies the specification.
Now we use the annotation method to prove the well-typing. We consider the following annotation:

Inherited assertion for plus: integer(second argument)
Synthesized assertion for plus: integer(first argument), integer(third argument).
No inherited assertion for plength.
Synthesized assertion for plength: list(first argument), integer(second argument).

The proof is performed showing locally in each clause how the type informations are propagated. Figure 1 illustrates the way to propagate the information “integer” or “list” inside the clauses. As there is no possible cycle following the arrows in any proof tree, the annotation is valid, i.e. the proof trees of roots “plus” whose second argument is an integer are of type integer and the proof trees of root “plength” are well typed (there is no inherited hypothesis attached to “plength”)

Note that our definition of the specifications, like our examples, does not restrict the domains of the interpretations to be Herbrand bases (we call them term bases). This will permit us to establish general results holding not only for pure logic programming but also for extensions like functional or constraint logic programming.

The paper is organized as follows. Section 1 introduces the basic definitions and notations. Special emphasis is given to the term (or Herbrand) interpretations, which corresponds to the most usual way to deal with the semantics of logic programs.
Section 2 introduces the definite clause programs, their semantics and the specifications. Two notions of validity are defined in order to deal with general interpretations, not with Herbrand’s ones only. This will permit us to obtain very general results on the proof methods. Section 3 presents the inductive proof method and explores its properties. Results of completeness, relative completeness (in the sense of [16]) and incompleteness are established.

Relationships with fixpoint and structural induction are investigated as the extension of the method to general programs and other kind of properties. Section 4 presents the proof method with annotations which can be viewed as a generalization of the inductive method but is not more powerful (it does not permit one to prove move valid specifications). In contrast, it permits one to prove properties holding inside the proof trees. In Section 5 a (short) example is fully developed with the purpose of showing the originality and the usefulness of the annotation method. Finally, a comparison with other results published in this literature is provided in Section 6.

1. Basic definitions and notations

1.1. Sort, signatures, terms

Context-free grammars and logical languages will be defined in the algebraic style of [18]. Some definitions are also useful to describe logic programs.

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

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

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

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

$$A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s \text{ if } f \text{ has profile } \langle s_1 \ldots s_n, s \rangle.$$

Let $V$ be an $S$-sorted set of variables (each $v$ in $V$ has arity $\epsilon$, 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 \( T(F) \) is considered as denoting a value \( t_A \) in \( A \), for an \( F \)-algebra \( A \). For any \( F \)-algebra \( A \) and an \( S \)-sorted set of variables, an assignment of values in \( A \) to variables \( V_s \), for all \( s \) in \( S \), is an \( S \)-indexed family of functions:

\[
v = \{ v_s : V_s \rightarrow A_s \}_{s \in S}.
\]

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

\[
v' = \{ v'_s : T(F, V) \rightarrow A_s \}_{s \in S}.
\]

In \( T \), assignments are called substitutions. For any assignment \( v \) in \( T \) and term \( t \) in \( T(F, V) \), \( v(t) \) is called an instance of \( t \).

### 1.2. Grammars

Proof trees of a logic program (see Definition 2.5) 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 \( \langle N, P \rangle \) where \( N \) is a finite set (the nonterminal alphabet) and \( P \) an \( N \)-sorted signature (for more details see [27]).

### 1.3. Many-sorted logical languages

The specifications 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 \( \text{bool} \) of the boolean values \( \text{true}, \text{false} \). Let \( V \) be a sorted set of variables, \( F \) an \( S \)-signature and \( R \) a finite set of many-sorted relation symbols (i.e. a set of symbols, each of them having an arity and, implicitly, the sort \( \text{bool} \)).

A logical language \( L \) over \( V, F, R \) is the set of formulas written with \( V, F, R \) and logical connectives like \( \forall, \exists, \Rightarrow, \land, \lor, \sim, \ldots \). We denote by \( \text{free}(\phi) \) the possibly empty set of the free variables of the formula \( \phi \) of \( L \) (\( \text{free}(\phi) \subseteq V \)), by \( \land \mathbb{N} \mathbb{D} A \) (resp. \( \lor \mathbb{N} \mathbb{D} A \)) the conjunction (resp. the disjunction) of formulas (\( \land \mathbb{N} \mathbb{D} \emptyset = \text{true} \), \( \lor \mathbb{N} \mathbb{D} \emptyset = \text{false} \)), and by \( \phi[u_1/v_1, \ldots, u_n/v_n] \) the result of the substitution of \( u_i \) for each free occurrence of \( v_i \), or \( \phi[u_1, \ldots, u_n] \). We do not restrict a priori the logical language to be first-order.

Let \( C(L) \) denote a class of \( L \)-structures or \( L \)-interpretations, i.e. objects of the form

\[
\mathbb{D} = \langle \{ D_s \}_{s \in S}, \{ f_D \}_{f \in F}, \{ r_D \}_{r \in R} \rangle.
\]

where \( \langle \{ D_s \}_{s \in S}, \{ f_D \}_{f \in F} \rangle \) is a heterogeneous \( F \)-algebra and for each \( r \) in \( R \), \( r_D \) is a total mapping

\[
D_{s_1} \times \cdots \times D_{s_n} \rightarrow \{ \text{true}, \text{false} \} = \text{bool} \quad \text{if} \quad \pi(r) = s_1 \ldots s_n.
\]

The notion of validity is defined in the usual way. For every assignment \( v \), every \( \mathbb{D} \) in \( C(L) \), every \( \phi \) in \( L \), one assumes that \( (\mathbb{D}, v) \models \phi \) either holds or does not hold. We say \( \phi \) is valid in \( \mathbb{D} \) and write \( \mathbb{D} \models \phi \) iff \( (\mathbb{D}, v) \models \phi \) for every assignment \( v \). In the sequel, we
will make extensive use of a distinction which is usual in the field of logic programming [44]. We will distinguish the algebraic part of an $L$-interpretation, called an $L$-preinterpretation ($L$ may be omitted if there is no ambiguity). It is the interpretation of the function symbols of $L$. Let us call $J$ an ($L$-) preinterpretation. Then $\mathbb{D}$ is a $J$-based $L$-interpretation iff $\mathbb{D}$ is the $L$-interpretation consisting of $J$ and the interpretation of the relations $R$ of $L$.

In the rest of the paper, the many-sorted approach will be used in most of the examples. This simplifies them. On the other hand, the presentation of the theory is simplified if a one-sorted approach is used. This will be done in all the theoretical sections. It is straightforward to extend the results obtained with the one-sorted approach to the many-sorted approach by quantifying all the variables in all the formulas over their corresponding domain. It is sufficient to assume that programs and formulas are well typed according to types and arities.

1.4. Examples

Here are some examples of interpretations for some languages.

**Integers language**

$$S = \{\text{int, pint, bool}\}$$

$L$ is

- $V$: variables
- $F = \{\text{zero, s, p, +}\}$
- $R = \{\text{plus, =}\}$

**Interpretation $\mathbb{N}$ (natural integers)**

- **Domains:** $N_{\text{int}} = \text{nat}$ (nat = posnat $\cup \{0\}$)
  
  $N_{\text{pint}} = \text{posnat}$ (posnat = $\{1, 2, \ldots\}$)
  
  $N_{\text{bool}} = \text{bool}$

- **Functions:** $\text{zero}_N \in \langle e, \text{nat} \rangle$: 0
  
  $s_N \in \langle \text{nat, posnat} \rangle: n \rightarrow n + 1$
  
  $p_N \in \langle \text{posnat, nat} \rangle: n \rightarrow n - 1$
  
  $+_N \in \langle \text{nat nat, nat} \rangle: (n, m) \rightarrow n + m$

- **Relations:** $=_N \in \langle \text{nat, nat, bool} \rangle: (n, m) \rightarrow \text{true}$ iff $n = m$
  
  $\text{plus}_N \in \langle \text{nat nat nat bool} \rangle: (n, m, l) \rightarrow \text{true}$ iff $l = n + m$

**Interpretation $\mathbb{E}$ (nonstandard signed integers) [Bid 81]**
Domains:

\[ E_{\text{int}} = \{(n, s) | n \in \text{nat}, s \in \{\pm 1\}\} \]
\[ E_{\text{posint}} = \{(n, s) | n \in \text{posnat}, s \in \{\pm 1\}\} \]
\[ E_{\text{bool}} = \text{bool} \]

Functions:

\[ \text{zero} \in \langle E_{\text{int}}, E_{\text{int}} \rangle : (0, +1) \]
\[ s_E \in \langle E_{\text{int}}, E_{\text{posint}} \rangle : (n, s) \rightarrow (n + 1, s) \]
\[ p_E \in \langle E_{\text{posint}}, E_{\text{int}} \rangle : (n, s) \rightarrow (n - 1, s) \]
\[ +_E \in \langle E_{\text{int}}, E_{\text{int}}, E_{\text{int}} \rangle : (n, s_1)(m, s_2) \rightarrow (n + m, s_2) \]

Relations:

\[ \in \in \langle E_{\text{int}}, E_{\text{int}}, \text{bool} \rangle : (n, s_1)(m, s_2) \rightarrow \text{true} \iff n - m \text{ and } s_1 = s_2 \]
\[ \text{plus} \in \langle E_{\text{int}}, E_{\text{int}}, \text{bool} \rangle : \]
\[ \text{plus}((n, s_1), (m, s_2), (l, s_3)) \rightarrow \text{true} \iff l = n + m \text{ and } s_3 = s_2. \]

Note that \( \mathbb{N} \) and \( E \) are models of the axioms

1. \( \text{plus}(\text{zero}, X, X) \)
2. \( \text{plus}(sX, Y, Z) \rightarrow \text{plus}(X, Y, Z) \)

i.e. these formulas are valid in \( \mathbb{N} \) and \( E \). However, the formula

3. \( \text{plus}(X, Y, Z) \Rightarrow \text{plus}(Y, X, Z) \) (commutativity)

is valid in \( \mathbb{N} \) but not in \( E \) (the commutativity is not a logical consequence of axioms (1) and (2), i.e. it does not hold in all models of (1) and (2)).

Algebraic language

Given a set of sorts \( S \) increased with \( \{\text{bool}, \text{int}\} \) and a language \( L \) over \( V, F, R \) as in \( T \) (Section 1.1) in which \( F \) contains also \( \{\text{size}\} \) and \( R \{\text{ground, - , var}\} \).

Interpretation \( T \) (canonical term interpretation)

Domains:

\[ T(F, V)_s \text{ set of the well-formed terms of sort } s \text{ following sorts and arities, built as follows:} \]
\[ \text{if } 1 \leq i \leq n, t_i \in T(F, V)_s \text{ and } f \in \langle s_1 \ldots s_n, s \rangle \]
\[ \text{then } f(t_1, \ldots, t_n) \in T(F, V)_s \]
\[ T_{\text{bool}} = \text{bool} \]
\[ T_{\text{int}} = \mathbb{N}_{\text{int}} \]

Functions:

\[ f_T \in \langle T(F, V)_s \ldots T(F, V)_s, T(F, V)_s \rangle : f \iff f \in \langle s_1 \ldots s_n, s \rangle \]
\[ \text{size}_T \in \langle T(F), T_{\text{int}} \rangle \text{ is defined recursively as follows:} \]
\[ \text{size}_T(f) = 1 \text{ if } f \text{ has arity 0 (constant) and } \text{size}_T(f(t_1, \ldots, t_n)) \]
\[ = 1 + \sum_{1 \leq i \leq n} \text{size}_T(t_i). \]
Relations:

\[ \text{ground}_T: \langle T(F, V), \text{bool} \rangle \mapsto \text{true} \iff \text{t is a term without variable} \]

\[ \text{size}_T: \langle T(F, V), \text{nat} \rangle \mapsto \text{size} \]

Note that ground \(_T\) and size \(_T\) are polymorphic and that size \(_T\) is a partial function on \(T(F, V)\).

\(\varepsilon, =_E\) and \(=_\varepsilon\) satisfy the congruence axioms of the equality as given in [44].

For example, the formula \((X_1 =_E Y_1 \land X_2 =_E Y_2) \Rightarrow (X_1 +_E X_2 =_E Y_1 +_E Y_2)\) is valid in \(\varepsilon\).

The following formula is valid in \(\varepsilon\).

\[ X =_T Y \Rightarrow \text{size}_T(X) =_\varepsilon \text{size}_T(Y) \]

If there is no ambiguity following the typing convention of the operators, subscripts denoting the models will be omitted.

Lists language

\[ S = \{\text{list, d-list, any, int, } T, \text{ boolean} \} \]

\[ F = \{[ ], \ldots, \text{nil}, \ldots, \text{append, } -, \text{repr, } \text{length} \} \]

\([ ]\) and \text{nil} are constants, \text{repr} has arity 1, the others arity 2.

\[ R = \{\text{is-a-list, is-a-dlist, permut, } = \} \]

**Interpretation** \(\varepsilon\) (lists and difference-lists)

Domains:

\[ L_{\text{list}} = \text{lists as usual (defined by 'nil' and '. ' denoting 'cons')} \]

\[ L_{\text{d-list}} = \text{d-lists (difference-lists as usual [50])} \]

\[ L_{\text{any}} = \text{elem (type of the lists elements: no restriction)} \]

\[ L_{\text{int}} = \text{nat (as in } \mathbb{N}) \]

\[ L_T = \text{is } T(F, V) \text{ as in } \varepsilon \]

\[ L_{\text{bool}} = \text{bool} \]

Functions:

\[ [ ]_{L} \in \langle \varepsilon, \text{ lists} \rangle \]

\[ \text{nil}_{L} \in \langle \varepsilon, \text{ lists} \rangle \]

\[ [ - ]_{L} \in \langle \text{elem lists, lists} \rangle \]

\[ -_{L} \in \langle \text{elem lists, lists} \rangle \]

\[ \text{append}_{L} \in \langle \text{lists, lists, lists} \rangle \]: concatenation of lists. "append\(_L\)" operator will be omitted in long formulas if there is no ambiguity on the type of the arguments.

\[ -_{L} \in \langle \text{lists, d-lists} \rangle \]: difference-lists constructor

\[ \text{repr}_{L} \in \langle \text{d-lists, lists} \rangle \]: defines the list represented by a difference-list

\[ \text{length}_{L} \in \langle \text{lists, nat} \rangle \]: the length of a list
Proof methods of declarative properties of definite programs

Relations:

- `is-a-list ∈ ⟨T(F, V), bool⟩`: `is-a-list(l)` is true iff `l` is a list built with list constructors.
- `is-a-d_list ∈ ⟨T(F, V), bool⟩`: `is-a-d_list(dl)` is true iff all instances of `dl` are `d_list` built with `d_list` constructors and represent a list (i.e. `repr(dl)` is defined). (For example, `is-a-d_list([X, Y | L] - [Y | L])` is true but `is-a-d_list([X, Y | L] - [Z | L])` is false.)
- `permut ∈ ⟨lists lists, bool⟩`: `permut(l1, l2)` is true iff `l1` is a permutation of `l2`.
- `= ∈ ⟨lists lists, bool⟩`: `l1 = l2` is true iff `l1` and `l2` are element-by-element identical lists (using `elem`).

1.5. Term interpretations

A special attention should be given to the term interpretations. They play a central role in logic programming as they are used to define the proof-theoretic and the fixpoint semantics of logic programs and as such they are also related to the computability of the relations defined by a logic program.

The basic idea of a term interpretation leads to its term domains: every value is denoted by a specific term (different from the others). Usually, term interpretations are one-sorted but they can be many-sorted also, as in the algebraic language example of Section 1.4.

The term domain is `T(F, V)`. It is closed by substitution. If `V` is empty (no variable) the term domain is ground and is also called the Herbrand universe. A term base `B` (or Herbrand base, if there is no variable) is the set of atoms `r(t1, . . . , tn)` with `r` in `R` and the `ti`’s in `T(F, V)`. For more details on term bases see [34, 29].

A term interpretation is defined by the canonical term preinterpretation in which the symbols are interpreted by themselves and, for each `r` in `R`, by a subset of `B` defining `r`.

By definition of a term base, to define a term interpretation `T` is just to define `r_T` for every `r` in `R`. Then one can identify the assignment in `T` with the substitutions ranging over terms and, moreover, for any assignment `v`, the value of a term by `v` is `tv` (`v` applied to `t`) and the notion of validity: `(T, v) |= r(t1, . . . , tn)` is equivalent to `(t1v, . . . , tnv) ∈ r_T`.

A term model of a set of formulas `S` is a term interpretation which satisfies all the formulas in `S`. A ground term model is also called a Herbrand model.

In an interpretation `D` the values of the domains are assumed to be different. Thus, in a term interpretation all terms which are not formally equal are different. This can be axiomatized by the following axioms which we will denote as the term-equality axioms (EQ). We borrow them from [2]:

1. (forall (x = x)).
2. (forall (f(x1, . . . , xn) = f(y1, . . . , yn) if x1 = y1 and . . . xn = yn)) for all function symbols `f`. 

(3) \(\forall (p(y_1, \ldots, y_n) \text{ if } p(x_1, \ldots, x_n)) \text{ if } x_1 = y_1 \text{ and } \ldots \text{ and } x_n = y_n\) for all predicates \(p\) including \(=\).

(4) \(\forall (x_1 = y_1 \text{ and } \ldots \text{ and } x_n = y_n \text{ if } f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))\) for all function symbols \(f\).

(5) \(\forall (f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_n))\) for all pairs of different function or constant symbols.

(6) \(\forall (t[x] \neq x)\) for all terms \(t[x]\) containing the variable \(x\) and different from \(x\).

Note that in every interpretation \(\mathcal{I}\) containing an interpretation of the equality which satisfies these axioms, if a value is represented by a term, this term is unique. In other terms, this means that all constants and functions are considered as constructors of different values, hence there is no axiom on constructors (initial algebra approach). This point will not be developed any more here.

2. Definite clause programs, specifications

2.1. Definition (definite clause program). A definite clause program (DCP) is a triple \(P = \langle \text{PRED}, \text{FUNC}, \text{CLAUS} \rangle\) where \(\text{PRED}\) is a finite set of predicate symbols, \(\text{FUNC}\) a finite set of function symbols disjoint of \(\text{PRED}\), \(\text{CLAUS}\) a finite set of clauses defined as usual \([12, 44]\) with \(\text{PRED}\) and \(\text{TERM} = T(\text{FUNC}, \mathcal{V})\). 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. Clauses are built as usual with \(\text{PRED}\) and \(\text{TERM}\).

2.2. Example (Program “plus”)

\[
\begin{align*}
\text{PRED} &= \{\text{plus}\} \quad \rho(\text{plus}) = 3 \\
\text{FUNC} &= \{\text{zero}, \text{s}\} \quad \rho(\text{zero}) = 0, \rho(\text{s}) = 1 \\
\text{CLAUS} &= \{c_1 : \text{plus}(\text{zero}, X, X) \leftarrow, \ c_2 : \text{plus}(X, Y, Z) \leftarrow \text{plus}(X, Y, Z)\}
\end{align*}
\]

Variables begin with uppercase letters.

2.3. Example (Program “permutations”). “List” terms are represented in Edinburgh syntax:

\[
\begin{align*}
\text{PRED} &= \{\text{perm}, \text{extract}\} \quad \rho(\text{perm}) = 2, \rho(\text{extract}) = 3 \\
\text{FUNC} &= \{[, ], [\cdots]\} \quad \rho([\ ])=0, \rho([\cdots])=2 \\
\text{CLAUS} &= \{c_1 : \text{perm} ([\ ], [\ ]\bigr) \leftarrow, \\
& \hspace{1cm} c_2 : \text{perm}(\langle A \mid L \rangle, \langle B \mid M \rangle \bigr) \leftarrow \text{perm}(N, M), \text{extract}(\langle A \mid L \rangle, B, N), \\
& \hspace{1cm} c_3 : \text{extract}(\langle A \mid L \rangle, A, L) \leftarrow, \\
& \hspace{1cm} c_4 : \text{extract}(\langle A \mid L \rangle, B, \langle A \mid M \rangle \bigr) \leftarrow \text{extract}(L, B, M)\}
\end{align*}
\]
2.4. Definition (Denotation of a DCP $P$: $\text{DEN}(P)$). The denotation of a DCP is the set of all its (not necessarily ground) atomic logical consequences:

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

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 $\text{DEN}(P)$ are valid in the models of $P$), since in this paper we will not make use of the logical semantics of a logic program, but rather of its proof-theoretic semantics that we shall now define. Other details can be found in \cite{12, 4, 44, 34, 29}.

2.5. Definition (J-based proof tree, proof tree). Given a DCP $P = \langle \text{PRED}, \text{FUNC}, \text{CLAUS} \rangle$ and a preinterpretation $J$ s.t. $J$ interprets all $\text{FUNC}$ in $P$ then the set of $J$-based proof trees of $P$ is defined as follows.

1. If $A$ is the result of the interpretation in $J$ of the arguments of a fact in $\text{CLAUS}$ whose variables (if any) have been assigned some value, then the tree consisting of one vertex with label $A$ is a $J$-based proof tree.

2. If $T_1, \ldots, T_q$ for some $q > 0$ are $J$-based proof trees with roots labelled $B_1, \ldots, B_q$ and if $A \leftarrow B_1, \ldots, B_q$ is the result of the interpretation in $J$ of all the arguments of the atoms of some clause in $\text{CLAUS}$ whose variables (in any) have been assigned some value, then the tree consisting of the root labelled with $A$ and the subtrees $T_1, \ldots, T_q$ is a $J$-based proof tree.

Intuitively, if we call $J$-based instances of a clause the result of assigning values to the variables of a clause and interpreting in $J$ the term arguments, it is easy to observe that a $J$-based proof tree is built by pasting together $J$-based instances of clauses, such that the leaves are $J$-based instances of facts. If this last condition is removed, one gets the notion of partial $J$-based proof tree.

Given an interpretation $\mathcal{D}$ of a class of $L$-structures such that $L$ includes $\text{FUNC}$ but not $\text{PRED}$, we denote by $\text{PTR}_{\mathcal{D}}(P)$ the set of all the proof tree roots (root labels) of the $J$-based proof trees of $P$, in which $J$ is the algebraic part of $\mathcal{D}$. If there is no confusion, we will speak of the $\mathcal{D}$-based proof trees instead of $J$-based. Analogously, we denote by $\text{PT}_{\mathcal{D}}(P)$ the set of the $\mathcal{D}$-based proof trees.

If $J$ is a term preinterpretation, then one finds the usual notion of (not necessarily ground) proof trees \cite{12, 34, 29}. Note that every instance of a proof tree is a proof tree also. The name “proof tree” will be reserved to denote term-based proof trees. In this case $\text{PTR}_{\mathcal{T}}(P)$ is just denoted as $\text{PRT}(P)$.

2.6. Proposition (Clark \cite{12}) (Proof theoretic semantics). Given a DCP $P$: $\text{DEN}(P) = \text{PTR}(P)$.

Thus, instead of the logical semantics of a logic program, one can deal with its proof-theoretic semantics. As pointed out in \cite{27}, proof trees can be thought of as syntax trees (terms of a clause algebra) “decorated” by atoms as specified in the proof
tree definition. Thus, inductive proof methods as defined in [18] may be applied to logic programs. This will be done in Section 3. Note that the set of all the D-based instances of the elements of DEN(P) — let us denote it DEN(P)_D — does not correspond in general to PTR_D(P) but is included in. They are the same if D is a term interpretation or if, in D, all values are represented by a term and all the axioms of EQ are valid. The consequence of the difference in the general case will be studied in Section 3.11.

It may be useful to observe that PTR_D(P) defines, with D, an interpretation which is a model of P. In particular, if D is a term-based interpretation, DEN(P) defines a model of P [12].

2.7. Definition (Specification of a logic program). A specification of a logic program P is a family of formulas $\mathcal{S} = \{\mathcal{S}_p\}_{p \in \text{PRED}}$ of a logical language L over V, F, R such that V contains the variables used in P and F contains FUNC, together with an L-structure D. For every p of PRED, we denote by varg(p) = \{p_1, \ldots , p_{\rho(p)}\} the set of variable names denoting any possible term in place of the 1st, \ldots , or \rho(p)th argument of p. Thus, we impose the condition free($\mathcal{S}_p$) $\subseteq$ varg(p).

For a good understanding of the results presented in this paper, it is important to remark that the relations R in the language L may or may not include PRED. There are two cases: L may contain PRED and a specification $\mathcal{S}$ may use these predicates. Moreover, the family of formulas may be reduced to the form $\mathcal{S}_p$: \( p(p_1, \ldots , p_n) \). In this case D can be viewed as a preinterpretation for P, augmented with the interpretations of the relations in R \( \cap \) PRED. In the second case — R does not contain any element of PRED — D is an interpretation of L and can simply be viewed as a preinterpretation for P. Without loss of generality, unless explicitly stated, it will be assumed in the sequel that the specification language L does not contain the predicate symbols of P. A specification $\mathcal{S}$ may also be viewed as defining an interpretation of the predicates in PRED as: \( p(v_1, \ldots , v_n) \in D \) iff \( (D, (v_1, \ldots , v_n)) \models \mathcal{S}_p \). We will denote by $I_{\mathcal{S}}$ the induced interpretation defined by a specification $\mathcal{S}$ (I_{\mathcal{S}} is the union of the $I_p$ for every p in PRED) and $\mathcal{S}_{P,D}$ the family of formulas such that $I_{\mathcal{S}_{P,D}} = \text{PTR}_D(P)$. One can assume that such formulas always exist (this will be discussed in Section 3.10). Given D, a preinterpretation for P, and the formulas $\mathcal{S}$, D augmented by $I_{\mathcal{S}}$ will be denoted as $D_{\mathcal{S}}$. $D_{\mathcal{S}}$ is a model of P if in particular $\mathcal{S} = \mathcal{S}_{P,D}$.

2.8. Definition (Valid specification, computational validity). A specification $\mathcal{S}$ on (L, D) is valid for the DCP P (or P is correct w.r.t. $\mathcal{S}$) iff

$$\forall p(v_1, \ldots , v_n) \in \text{PTR}_D(P), \quad D \models \mathcal{S}_p[v_1/p_1, \ldots , v_n/p_n], \quad n = \rho(p).$$

In practice, there is a more interesting definition of a “valid specification” which we will call “computational correctness” if we need to distinguish it from the previous one.
A specification $P$ on $(L, D)$ is computationally valid (or $P$ is computationally correct w.r.t. $\mathcal{S}$) iff

$$\forall p(t_1, \ldots, t_n) \in \text{DEN}(P) \quad D \vdash \mathcal{S}[t_1, \ldots, t_n].$$

This definition has been used in [20]. Without the precautions taken here, the completeness of the method stated in Theorem 3.5 does not hold. Nevertheless, with any kind of interpretation $D$, if a specification is valid in the sense of the $D$-based proof trees, it is also valid in the sense of DEN as $\text{DEN}(P)_D \subseteq \text{PTR}(P)$. In other words, if a specification is valid for a DCP, it is also valid computationally.

The latter definition above means that every atom of the denotation satisfies the specification (with a universal quantification of the variables in the terms), hence every atom in any proof tree. It means also that every answer substitution (if any) satisfies the specification.

Both definitions correspond to a notion of partial correctness referring to the declarative (i.e. proof-theoretic or logical) semantics since nothing is specified about the existence of proof trees (the denotation can be empty), the way to compute them or the kind of resulting answer substitution for a given goal. If $\text{DEN}(P)_D$ and $\text{PTR}(P)_D$ are the same then both definitions coincide. Until Section 3.10 only the first definition will be used.

Note that any logical consequence of a valid specification is also valid, i.e. if $\mathcal{S}$ is valid and $D \models \mathcal{S} \Rightarrow \mathcal{S}'$ then $\mathcal{S}'$ is valid. ($\mathcal{S} \Rightarrow \mathcal{S}'$ stands as a shorthand for the family of implications $\mathcal{S} \Rightarrow \mathcal{S}^p$ for all $p$ in PRED).

2.9. Example (specification for Example 2.2)

$L_1 = V_1$ contains $\text{varg(plus)} = \{\text{plus1, plus2, plus3}\}$

$F_1 = \{\text{zero, s, +}\}$

$R_1 = \{=\}$

$D_1 = \mathbb{N}$ as in Section 1.4

$\mathcal{S}_1 = \{\mathcal{S}_1^{\text{plus}}\}, \quad \mathcal{S}_1^{\text{plus}}: \text{plus3 = plus1 + plus2}$

The validity of $\mathcal{S}_1$ (which is proved in Section 3) means that the program "plus" in Section 2.2 specifies the addition in $\mathbb{N}$, in particular that every $n$-tuple of values corresponding to the interpreted arguments of the elements of the denotation satisfies the specification $\text{plus3 = plus1 + plus2}$.

More precisely, it means that if the variables appearing in a proof tree are assigned over the domains of the functions or predicate in which they appear (i.e. here $N_{\text{int}}$), all the atoms at the nodes of the proof tree satisfy the corresponding specification if function symbols are interpreted as in $\mathbb{N}$. If one wants to make clear that this
specification holds for integer values only (if, for example, this program may be used in different contexts), one could use the following specification:

\[ \mathcal{S}_{\text{plus}}^{1} = (\text{integer}(\text{plus}2) \lor \text{integer}(\text{plus}3)) \Rightarrow \text{plus}3 = \text{plus}1 + \text{plus}2 \]

In this case it will be clear that plus1, plus2 and plus3 are always integers as the following specification is also valid:

\[ \mathcal{S}_{\text{plus}}^{2} : (\text{integer}(\text{plus}1) \land \text{integer}(\text{plus}2)) \Leftrightarrow \text{integer}(\text{plus}3) \]

Another interesting property may be

\[ L_{2} = V_{2} \text{ as in } L_{1}, \quad F_{2} = \{ \text{zero, s} \}, \]
\[ R_{3} = \{ \text{ground} \} \quad \rho(\text{ground}) = 1. \]
\[ D_{2} = \mathbb{T} \text{ as in Section 1.4} \]
\[ \mathcal{S}_{3} = \{ \mathcal{S}_{\text{plus}}^{2} \}, \quad \mathcal{S}_{\text{plus}}^{2} : (\text{ground}(\text{plus}3) \Rightarrow \text{ground}(\text{plus}2)) \land \text{ground}(\text{plus}1) \]

\( \mathcal{S}_{3} \) is a valid specification (it can be observed on every proof tree and will be proved in the next section).

2.10. **Example** (specification for Example 2.3). This example uses a many sorted \( L_{3} \) structure:

\[ L_{3} = V_{3} \text{ contains } \text{varg}(\text{perm}) \text{ and } \text{varg}(\text{extract}). \]
\[ F_{3} = \{ [ ], [ \_ \_ \_ ], \text{nil}, \ldots, \text{append} \ [ ] \text{ and nil are constants}, \]
\[ \text{the other operators have arity 2.} \]
\[ R_{3} = \{ \text{is-a-list, permut} \} \quad \rho(\text{is-a-list}) = 1, \rho(\text{permut}) = 2. \]
\[ D_{3} = \mathbb{L} \text{ as in Section 1.4} \]
\[ \mathcal{S}_{3} = \{ \mathcal{S}_{\text{perm}}, \mathcal{S}_{\text{extract}}, \exists L_{1}, L_{2} (\text{extract}1 = \text{append}(L_{1}, \text{extract}2), \text{extract3} = \text{append}(L_{1}, L_{2})) \]

3. **Inductive proof method**

3.1. **Definition** (Inductive specification \( \mathcal{S} \) of a DCP \( P \)). A specification \( \mathcal{S} \) on \( (L, D) \) of a DCP \( P \) is inductive iff for every \( c \) in \( \text{CLAUS}, \)

\[ c : p_{0}(t_{01}, \ldots, t_{0_{n_{0}}}) \leftarrow p_{1}(t_{11}, \ldots, t_{1_{n_{1}}}), \ldots, p_{m}(t_{m1}, \ldots, t_{m_{m_{m}}}), \]

\[ D \models (\forall \mathbb{N} \mathbb{D} \mathcal{S}^{p}[t_{k_{1}}, \ldots, t_{k_{nk}}] \{ \mathcal{S}^{p}[t_{01}, \ldots, t_{0_{n_{0}}}] \}) \quad (1) \]

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. Remark that in (1) the remaining variables are variables of the clause \( c \) and they are universally quantified over their domains defined in \( D \).
3.2. Proposition (An inductive specification is valid). If a specification \( \mathcal{S} \) of \( P \) is inductive then \( \mathcal{S} \) is valid for \( P \).

**Proof.** By an easy induction on the size of \( D \)-based proof trees, if \( \text{PTR}_{D,n}(P) \) denotes the set of all the roots of the proof tree of size \( \leq n \), \( \mathcal{S} \) holds in \( \text{PTR}_{D,n+1}(P) \) (by Definitions 2.5 and 3.1 and the notion of validity), thus in \( \text{PTR}_{D}(P) = \cup \text{PTR}_{D,n}(P) \).

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

We denote by \( \mathcal{S}_{\text{true}} \) the specification such that \( \mathcal{S}^p \) is true for all \( p \) in \( \text{PRED} \) (i.e. no specification).

3.4. Proposition (The strongest specification is inductive). Given a DCP \( P \), \( \mathcal{S}_P, D \) and \( \mathcal{S}_{\text{true}} \) are, respectively, the strongest and the weakest valid specification for \( P \), and \( \mathcal{S}_{P,D} \) is inductive, i.e. all valid specifications \( \mathcal{S} \) satisfy:

\[
D \models (\mathcal{S}_{P,D} \Rightarrow \mathcal{S} \Rightarrow \mathcal{S}_{\text{true}}) \quad \text{and} \quad \mathcal{S}_{P,D} \text{ is inductive.}
\]

**Proof.** It is easy to observe that \( \mathcal{S}_{P,D} \) is inductive as it corresponds, by definition, to the interpretation defined by \( \text{PTR}_{D}(P) \) whose elements are \( D \)-based proof tree roots (i.e. built with \( D \)-based instances of clauses). On the other hand, every valid specification is, by definition, true for every \( n \)-tuple of values of a \( D \)-based proof tree root. Hence \( D \models \mathcal{S}_{P,D} \Rightarrow \mathcal{S} \). Obviously, \( D \models \mathcal{S} \Rightarrow \mathcal{S}_{\text{true}} \).

3.5. Theorem (soundness and completeness of the inductive proof method). A specification \( \mathcal{S} \) on \( (L, D) \) is valid for \( P \) if it is weaker than some inductive specification \( \mathcal{S}' \) on \( (L, D) \), i.e. \( \exists \mathcal{S}' \) such that

1. \( \mathcal{S}' \) inductive,
2. \( D \models \mathcal{S}' \Rightarrow \mathcal{S} \).

**Proof.** Soundness is trivial since, if \( \mathcal{S}' \) is inductive, it is valid by Proposition 3.2 and, if \( D \models \mathcal{S}' \Rightarrow \mathcal{S} \), then \( \mathcal{S} \) is valid by the remarks following Definition 2.8.

Completeness results from Proposition 3.4 with \( \mathcal{S}' = \mathcal{S}_{P,D} \). Note that one does not have the completeness if one restricts the specification language to be first-order (see Section 3.10).

3.6. Example (The specification \( \mathcal{S}_1 \) (Example 2.9) is inductive). Following Definition 3.1, it is sufficient to prove that

\[
\mathbb{N} \models \mathcal{S}_1 [\text{zero}, X, X] \quad \text{and} \quad \mathbb{N} \models \mathcal{S}_1 [X, Y, Z] \Rightarrow \mathcal{S}_1 [s(X), Y, s(Y)]
\]
or

\[
\mathbb{N} \models 0 + X = X \quad \text{and} \quad \mathbb{N} \models (X + Y = Z \Rightarrow X + 1 + Y = Z + 1)
\]

which are valid formulas in \(\mathbb{N}\).

### 3.7. Example
(The specification \(\mathcal{S}_2\) (Example 2.9) is inductive). In the same way, it is easy to show that the following formulas are valid on \(\mathcal{D}_2\):

\[
\text{ground}(\text{zero}) \land (\text{ground}(X) \Rightarrow \text{ground}(X))
\]

and

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

### 3.8. Example
(The specification \(\mathcal{S}_3\) (Example 2.10) is inductive). It is easy to show that the following formulas are valid on \(\mathcal{D}_3\) (some replacements are already made in the formulas and universal quantifications on the variables are implicit):

in \(c_1\):

permut(nil, nil)

in \(c_2\):

\[
\text{permut}(N, M) \land \exists L_1, L_2 \ (A. L = \text{append}(L_1, B. L_2) \land N = \text{append}(L_1, L_2)) \Rightarrow \text{permut}(A. L, B. M)
\]

\(L_1\) and \(L_2\) are lists) take \(L_1 = \text{nil}\) and \(L_2 = L\).

in \(c_3\):

\[
\exists L_1, L_2 \ (A. L = \text{append}(L_1, B. L_2) \land L = \text{append}(L_1, L_2)) \Rightarrow \exists L_1', L_2' \ (A. L = \text{append}(L_1', B. L_2') \land A. M = \text{append}(L_1', L_2'))
\]

take \(L_1' = A. L_1\) and \(L_2' = L_2\).

### 3.9. More examples
We achieve this illustration with some more examples of inductive proofs.

**Concatenation**

We define the concatenation of difference lists as usual by one fact:

\[
\text{concat}(L_1 - L_2, L_2 - L_3, L_1 - L_3) \leftarrow
\]

Note that in this example we introduce explicitly the typing predicates as “repr” is a partial function over difference lists.

\[
\mathcal{S}_{\text{concatenate}} : \text{is-a-d_list} (\text{concatenate1}) \land \text{is-a-d_list} (\text{concatenate2})
\]

\[
\Rightarrow \text{repr}(\text{concatenate3}) = \text{append}(\text{repr}(\text{concatenate1}), \text{repr}(\text{concatenate2}))
\]

defined on \((L_3, \mathcal{D}_3)\) is inductive.

The claim is obvious, there is only one fact and

\[
\text{is-a-d_list}(L_1 - L_2) \land \text{is-a-d_list}(L_2 - L_3)
\]

\[
\Rightarrow \text{repr}(L_1 - L_3) = \text{append}(\text{repr}(L_1 - L_2), \text{repr}(L_2 - L_3))
\]
Graph colouring

We consider a graph colouring program whose (second-order) specification is the following:

$$\exists \mathbf{f} : \text{regions} \rightarrow \text{colours} \text{ such that } \forall r_i, r_j ~\text{regions}, \ r_i \neq r_j \land \text{adjacent}(r_i, r_j) \Rightarrow \mathbf{f}(r_i) \neq \mathbf{f}(r_j).$$

We denote by solution($\mathbf{f}$) the property required for the mapping $\mathbf{f}$.

The interpretation we are interested in consists of a description of pairs of different adjacent regions and different nonadjacent ones and of mappings which will be represented by lists of pairs $\langle \text{region}, \text{colour}\rangle$.

Here is a definite program whose purpose is to specify mappings satisfying $P._{\text{solution}}$.

$$c_1 : p._{\text{solution}}([ ]) \leftarrow$$
$$c_2 : p._{\text{solution}}([\langle R, C \rangle]) \leftarrow \text{colour}(C), \text{region}(R).$$
$$c_3 : p._{\text{solution}}([\langle R_1, C_1 \rangle, \langle R_2, C_2 \rangle|\mathcal{S}|]) \leftarrow p._{\text{adjacent}}(R_1, R_2),$$
$$\text{diff}_c(C_1, C_2),$$
$$p._{\text{solution}}([\langle R_1, C_1 \rangle|\mathcal{S}|]),$$
$$p._{\text{solution}}([\langle R_2, C_2 \rangle|\mathcal{S}|]).$$
$$c_4 : p._{\text{solution}}([\langle R_1, C_1 \rangle, \langle R_2, C_2 \rangle|\mathcal{S}|]) \leftarrow p._{\text{noadjacent}}(R_1, R_2),$$
$$p._{\text{solution}}([\langle R_1, C_1 \rangle|\mathcal{S}|]),$$
$$p._{\text{solution}}([\langle R_2, C_2 \rangle|\mathcal{S}|]).$$

The following specification is inductive, hence valid:

$$\mathcal{S} = \{ p._{\text{solution}} : \text{solution}($$ solution1 $) \land \text{in solution1 all pairs are different (no redundancy),}$$

$$\mathcal{S}_{\text{adjacent}} : \text{adjacent1} \land \text{adjacent2} \land \text{adjacent(adjacent1, adjacent2)} \land \text{adjacent1 and adjacent2 are regions,}$$

$$\mathcal{S}_{\text{noadjacent}} : \text{noadjacent1} \neq \text{noadjacent2} \land \sim \text{adjacent(noadjacent1, noadjacent2)} \land \text{noadjacent1 and noadjacent2 are regions,}$$

$$\mathcal{S}_{\text{diff}_c} : \text{diff}_c(C_1) \neq \text{diff}_c(C_2) \land \text{diff}_c(C_1) \land \text{diff}_c(C_2) \text{ are colours,}$$

$$\mathcal{S}_{\text{colour}} : \text{colour1 is a colour,}$$

$$\mathcal{S}_{\text{region}} : \text{region1 is a region}\}$$

One may assume that "$p._{\text{adjacent}}"", "p._{\text{noadjacent}}"" and "$\text{diff}_c"" are given by facts satisfying the corresponding specification (adjacent arguments are regions and $\text{diff}_c$ arguments are colours). It remains to prove that $\mathcal{S}$ is inductive in the clauses of "solution".
In $c_1$ and $c_2$, the result holds trivially.

In $c_3$, one may prove separately that

$$\text{solution}([\langle R_1, C_1 \rangle | S]) \land \text{solution}([\langle R_2, C_2 \rangle | S]) \land C_1 \neq C_2 \land R_1 \neq R_2 \land$$

$$\text{adjacent}(R_1, R_2)$$

$$\Rightarrow \text{solution}([\langle R_1, C_1 \rangle, \langle R_2, C_2 \rangle | S])$$

and

$$(\text{no redundancy in } [\langle R_1, C_1 \rangle | S] \land \text{no redundancy in } [\langle R_2, C_2 \rangle | S] \land$$

$$C_1 \neq C_2 \land \text{adjacent}(R_1, R_2)) \land R_1 \neq R_2)$$

$$\Rightarrow \text{no redundancy in } [\langle R_1, C_1 \rangle, \langle R_2, C_2 \rangle | S].$$

It is obvious.

In $c_4$, same kind of reasoning applies.

\section{3.10. Wand's incompleteness results}

Theorem 3.5, which states the completeness of the method, has been obtained assuming that the formulas $\mathcal{S}_{P,D}$ always exist. We now turn to the problem whether such formulas exist or not.

In [18] it is shown that Wand's incompleteness results established for Hoare's like deductive systems hold also for inductive proofs in attribute grammars; that is to say that the assertion language, if restricted to first-order (finite) formulas, may not be rich enough to express the (inductive) properties needed to achieve the proof of some specification.

For this reason we did not restrict our specification languages to be first-order and we pointed out (Definition 2.7) the need of a language large enough such that the result on the completeness of the method could be stated (Theorem 3.5).

However, one could suspect that such incompleteness results might not hold, as we are concerned with definite programs which are, as shown in [18], very particular attribute grammars. In other words, one could expect that starting from definite programs which are already first-order logical specifications, the inductive proof method could still remain complete in the framework of first-order logic.

Unfortunately, it is not the case. In other words, the formulas $\mathcal{S}_{P,N}$ on $(L, \mathbb{D})$ do not always exist if one restricts the language $L$ to be first-order. We give a formal proof of this result by re-coding Wand's counterexample [53, 18] in the definite program's style.

We show an example of definite program $P$ and a valid specification for $P$ defined on an $L$-structure $M$ such that no first-order inductive specification defined with $L$ can be found.

Let $L$ be the language defined as follows:

Let $p, q, r$ be unary predicates and $f$, a unary function symbol. It is assumed to be a predicate "=", which will be interpreted as the identity of the values of the domains.
Let \( \mathcal{M} \) be the first-order \( L \)-structure such that its domain in \( M \), with
\[
M = \{ a_n \mid n \geq 0 \} \cup \{ b_n \mid n \geq 0 \},
\]
\[f \in \langle M, M \rangle \text{ is } f(a_0) = a_0, f(b_0) = b_0, \]
\[f(a_n) = a_{n-1}, f(b_n) = b_{n-1} \text{ for } n \geq 1 \]
\[p(X) \text{ true iff } X = a_0 \]
\[q(X) \text{ true iff } X = b_0 \]
\[r(X) \text{ true iff } X = a_n \text{ for some } n \text{ of the form } k(k+1)/2. \]

It is shown in [53, Theorem 21] that there is no first-order formula \( \varphi \) with one free variable \( X \) such that for \( d \) in \( M \), \( (\mathcal{M}, d) \models \varphi \) iff \( d \in \{ a_n \mid n \geq 0 \} \).

Let \( P \) be the following pure logic program:
\[
\text{PRED} = \{ \text{rec} \}
\]
\[
\text{FUNC} = \{ f \}
\]
\[
\text{CLAUS} = \{ \text{rec}(X, Y) \leftarrow \text{rec}(f(X), Y), \text{rec}(X, X) \leftarrow \text{equal}(f(X), X), \text{equal}(X, X) \leftarrow \}
\]

It is easy to see that the strongest specification \( \mathcal{S}_{P,M} \) for \( P \) is the following (inductive, but we will show that \( \mathcal{S}_{\text{rec}} \) is not first-order expressible in \( L \)):
\[
\mathcal{S}_{\text{rec}}: \ (\text{rec1} \in \{ a_n \mid n \geq 0 \} \land \text{rec2} = a_0) \lor (\text{rec1} \in \{ b_n \mid n \geq 0 \} \land \text{rec2} = b_0)
\]
\[
\mathcal{S}_{\text{equal}}: \ \text{equal1} = \text{equal2}
\]

Let \( \Theta \) be the valid specification such that
\[
\Theta_{\text{rec}} = \text{r(rec1)} \Rightarrow \text{p(rec2)}
\]
\[
\Theta^X = \text{true for } X \in \{\text{equal}\}
\]

Assume there is an inductive specification \( \Theta_i \) such that \( \Theta_i \Rightarrow \Theta \). It must satisfy the following conditions:
1. \( \Theta_i^{\text{rec}}(\text{rec1}, \text{rec2}) \land \text{r(rec1)} \Rightarrow \text{p(rec2)} \)
2. \( \Theta_i^{\text{rec}}(f(X), Y) \Rightarrow \Theta_i^{\text{rec}}(X, Y) \)
3. \( f(X) = X \Rightarrow \Theta_i^{\text{rec}}(X, X) \) (\( \Theta_i^{\text{equal}} \) is \( \text{equal1} = \text{equal2} \))

Now let \( \varphi \) be the formula
\[
\varphi(x): \ \forall y \Theta_i^{\text{rec}}(x, y) \Rightarrow p(y)
\]

We will show that \( (\mathcal{M}, d) \models \varphi \) iff \( d \in \{ a_n \mid n \geq 0 \} \).

"If": If \( \Theta_i^{\text{rec}}(a_n, v), n \geq 0 \) holds then, by (2), \( \Theta_i^{\text{rec}}(a_m, v) \), for all \( m > n \). Let us choose \( m > n \) such that \( r(a_m) \) holds; thus, by (1), \( p(v) \) also holds.

"Only if": From (3), \( \Theta_i^{\text{rec}}(b_n, v) \), \( n \geq 0 \) holds; hence, by (2), \( \Theta_i^{\text{rec}}(b_n, b_0) \) for \( n \geq 0 \). Now take \( \Theta_i^{\text{rec}}(b_n, v) \), \( v = b_0 \); it holds, but, since \( p(b_0) \) does not, \( (\mathcal{M}, d) \models \varphi \) does not hold if \( d \notin \{ a_n \mid n \geq 0 \} \).

By Wand's result, \( \varphi \) cannot be first-order, hence \( \Theta_i^{\text{rec}} \) also cannot be.
Remark that in this example we used a nondeterministic coding of the deterministic one given in [53, 18].

3.11. The relative completeness of the proof method holds also with term interpretations

The result of relative completeness has been obtained with an example whose logic program has an empty denotation. It corresponds to a limit case in which $\text{DEN}(P)_D$ (here empty) and $\text{PTR}_D(P)$ are different, as illustrated in Fig. 2. If $\mathbb{D}$ is a term interpretation $\text{DEN}(P)_D$ and $\text{PTR}_D(P)$ are the same.

In this case, validity and computational validity coincide and every element of $\text{PTR}_D(P)$ is the root of a proof tree obtained with “instances” of clauses of $P$ in $\mathbb{D}$. Assume that $\mathbb{D}$ satisfies the equality axioms (EQ) and that every value in $\mathbb{D}$ can be represented by a term. Then, to every instance of a clause, these corresponds an instance in the term interpretation, hence the corresponding $\mathbb{D}$-based proof tree roots are also elements of $\text{DEN}(P)$. It follows that Theorem 3.5 gives a sound and complete method to prove the computational validity of specifications expressed on term interpretations.

By the proof above one could suspect that the incompleteness result originates in the possible differences between $\text{DEN}(P)_D$ and $\text{PTR}_D(P)$ due to unrestricted models and that the method could remain complete with first-order assertions in the case of term interpretations.

One could be convinced about this noting that in the Herbrand universe every value is uniquely described by a term and that all values are finitely represented. In the above example one could have represented all values of $M$ by constants indexed in $\mathbb{N}$; doing that, and with few modifications of the logic program, one is able to formulate the strongest inductive specification by a finite first-order formula.

We show now that even with pure Herbrand domains the incompleteness result still holds. Let us consider the assertion $\mathcal{F}_{P,D}$ in the case of a term interpretation. Every formula $\mathcal{F}_{P,D}$ can be expressed as a (usually infinite) disjunction of equalities:

$$\mathcal{F}_{P,D}: \bigvee \overline{p} = \overline{i}$$

$$p(\overline{i}) \in \text{DEN}(P)$$

![Fig. 2.](image-url)
Consider also a term interpretation in which the only predicate symbol is "\(=\)" which satisfies the axioms EQ. ("\(=\)" is interpreted as the term identity). The specification language \(L\) uses only the function symbols of the program \(P\) and the predicate symbol "\(=\)". Thus, we are faced with the following problem: does there exist a finite first-order formula equivalent to the (usually infinite) formula \(9\land F_n\)?

Assume there exists \(F^p\) equivalent to \(\mathcal{P}^p\). \(F^p[\bar{t}]\) is true iff \(p[\bar{t}]\) belongs to \(\text{DEN}(P)\). It is shown in [18] that there exists a decision procedure for the validity in the Herbrand universe of any first-order formula with the only predicate symbol "\(=\)". It follows that if \(F^p\) exists for all predicates \(p\) of \(P\), then \(\text{DEN}(P)\) is recursive (\(F\) decidable). But it is known that any Turing machine can be encoded by a definite program whose denotation contains only its halting states (see [27] for such a description). Hence the incompleteness results.

The incompleteness of the method does not come from the nature of the interpretation but, rather, is relative to the nature of the language, too limited if restricted to the first-order logic.

3.12. Definitions (IF, ONLY-IF, IFF, COMP axioms). First let us recall some classical definitions borrowed from [2]. Given a DCP \(P\), the following steps define the IF\((P)\), ONLY-IF\((P)\) and IFF\((P)\).

Step1: remove terms:
  Transform each clause \(p(t_1,\ldots,t_n) \leftarrow a_1,\ldots,a_m\) of \(P\) into \(p(x_1,\ldots,x_n) \Rightarrow x_1 = t_1 \land \cdots \land x_n = t_n \land a_1 \land \cdots \land a_m\).

Step2: Introduce existential quantifiers: Let \(y_1,\ldots,y_q\) be the variables of the original clause. Transform each formula \(p(x_1,\ldots,x_n) \leftarrow F\) into \(p(x_1,\ldots,x_n) \leftarrow \exists y_1,\ldots,y_q F\).

Step3: Group similar formulas: Let \(p(x_1,\ldots,x_n) \leftarrow F_1,\ldots,p(x_1,\ldots,x_n) \leftarrow F_k\) be all formulas obtained in the previous step with a relation \(p\) on the left-hand side. Replace them by one formula: \(p(x_1,\ldots,x_n) \leftarrow F_1 \lor \cdots \lor F_k\).

Step4: Handle "undefined" relation symbols:
  For each \(n\)-ary relation symbol \(q\) not appearing in a head of a clause in \(P\) add a formula, \(q(x_1,\ldots,x_n) \leftarrow \text{false}\).

Step5: Introduce universal quantifiers: IF axioms.
  Replace each formula \(F\) by \(\forall(F)\).

Step6: Add ONLY-IF axioms.
  For each formula \(\forall(a \leftarrow F)\) obtained at step 5 add a new formula \(\forall(a \Rightarrow f)\).

We call the intermediate form obtained after step 5 the IF definition associated with \(P\), denoted IF\((P)\), the set of formulas added at step 6 is the ONLY-IF definition associated with \(P\), denoted ONLY-IF\((P)\), and the union of both sets is the IFF definition, denoted IFF\((P)\).

The completion of a program \(P\) is the union of IFF\((P)\) and EQ (Section 1.5). It is denoted COMP\((P)\).
Given a set of formulas $F$, we will denote by $F[Y]$ the replacement of all the occurrences of the atoms $p(t_1, \ldots, t_n)$ in $F$ by $Y_p[t_1, \ldots, t_n]$, the formula associated with $p$ in the specification $Y$ in which every free variable $p_i$ corresponding to the $i$th argument is replaced by the term $t_i$ appearing as the $i$th argument of the occurrence of $p$ in $F$. It is important to observe that the models of $IF(P)$ are the same as the models of $P$ and that $D \models P[Y]$ is equivalent to saying that $I_Y$ defines, with $D$ taken as a preinterpretation for $P$, a model of $P$ (i.e. $D, \models P$).

3.13. A view on fixpoint induction

In [47] it is shown that the following proof rule (called fixpoint induction) is sound. We restate it in the logic programming framework:

(1) $D \models IF(P)[Y]$ then (2) $D \models \text{conv}(P) \Rightarrow Y$

where $\text{conv}(P)$ is the least fixpoint of the system (1) in the sense of Knaster–Tarski, i.e. the least solution $Y$ of (1) (which is also, by monotonicity, the least solution of $IFF(P)[Y]$ in $D$).

By the same fixpoint theorem it is known that

(3) $D \models Y_{P,D} \Rightarrow \text{conv}(P)$

(this point will be clarified below). Hence Theorem 3.5 (soundness) is proved again as (1) means that $Y$ is inductive and from (2) and (3) $Y$ is valid $D \models Y_{P,D} \Rightarrow Y$ is the definition of validity (Definition 2.8).

This shows how to obtain some extensions of the proof method. The fixpoint induction is sound as far as the system $IF(P)[Y]$ is monotonic in $Y$ and easy to use if $\text{conv}(P)$ may easily be characterized in terms of a valid specification.

The first point can be achieved if one considers logic programs built with clauses in which the body can be any first-order formula, but in which the predicates to which a specification may be attached are positive. This includes bodies without negated predicates, or constraint logic programming in which the negations are in the constraints only, clearly distinguished from the other predicates. Thus, as stated in [47, (3.2)], if the body of the clauses involves just $\forall, \exists, \land, (\sim \text{ in constraints only})$, then the monotonicity is preserved.

With regard to the second point it is known that if $P$ is a normal program (i.e. the bodies of the clauses may contain negated atoms) and if $P$ is stratified in the sense of [44, p. 110] then there exists a strongest valid specification satisfying (1) expressed in terms of a minimal model of the completion of $P$. But the system (1) is not always monotonic any more and (3) may not hold.

Combining both results one may expect to be able to use the same method for programs more general than the definite ones. One extension could be logic programs without negation but any first-order formula in the body. Another has been studied in [23, 24] for “well-founded” logic programs.

Another, but similar, way to look at fixpoints is to consider the $D$-based immediate consequence operator $T_{P,D}$ as defined in [44, 2] by:

$$T_{P,D}(I) = \{ a^0 \models \exists c: a_0 \leftarrow a_1, \ldots, a_m \text{ and } \exists v \text{ s.t. } a^0_i \models I, i > 0 \}$$
It is known that $I$ is a model of $\text{IF}(P)$ (or of $P$) iff $T_{P,0}(I) \subseteq I$ and of $\text{ONLY-IF}(P)$ iff $I \subseteq T_{P,0}(I)$ and of $\text{IFF}(P)$ iff $T_{P,0,B}(I) = I$. Hence the fixpoint induction can be restated as follows:

(1)' $T_{P,D}(I) \subseteq I$ then (2)' $\text{lfp}(P) \subseteq I$

in which $\text{lfp}(P)$ is the least fixpoint of $T_{P,D}$ which contains, by the monotonicity of $T_{P,D}$, the set of the $D$-based proof tree roots, namely $(\text{PTR}_D(P))$ is just $T_{P,D} \uparrow \omega)$:

(3)' $\text{PTR}_D(P) \subseteq \text{lfp}(P) \subseteq I$

This gives another formulation of the proof of Theorem 3.5 (soundness).

3.14. Proof method extends to amalgamation of DCPs and other programming

All the results expressed so far do not require the interpretations to satisfy the properties of term domains (equality axioms). This means that the proof method holds for something obviously more general than just usual DCPs but also for DCPs and functional programming (or many kinds of amalgamations).

As an example, consider the program FIB (Fibonacci):

FIB: fib(0, 1)←
    fib(1, 1)←
    fib(N, R1 + R2)← N > 1, fib(N - 1, R1), fib(N - 2, R2).

in which all the function symbols are interpreted on $\mathbb{N}$ (Section 1.4) (augmented with the predicate symbol $>$ interpreted as usual) and the constants are the values of nat.

One can show by the inductive method that the following specification is inductive (hence valid):

fib $\in$ nat $\land$ fib2 = fibonacci (fib 1)

This example shows also the possibility of applying the same proof method to programs with built-in predicates ($N > 1$ in this example is interpreted as "$>$" in $\mathbb{N}$).

3.15. An axiomatic view of the proof method

The proof method has been formulated so far using interpretations. This is not the most usual approach, especially if one wants to make automatized proofs. In this case it is assumed that a specification is expressed in a language whose meaning is given by a set of axioms, i.e. a subset of $L$ denoted Ax.

The definitions of validity and inductive specification need to be adapted considering that it is a kind of generalization from one interpretation to a class of interpretations (those which satisfy the axioms Ax). These interpretations will be referred to in the sequel as the interpretations of the axioms Ax. It should be clear that, due to the inclusion of all the function symbols of the program into the language of the specifications, the interpretations which we will consider are also preinterpretations for $P$. It will be assumed also that there is no axiom concerning the predicates of $P$ in
Ax and that the interpretations do not include the predicates of $P$. If $Ax$ contains another axiomatization of $P$, most of the definitions are simplified and this will be discussed in Section 6.

3.15.1. Definition (Valid specification (axiomatic view)). A specification $\mathcal{S}$ on $L$ with axioms $Ax$ is valid for the DCP $P$ (or $P$ is correct w.r.t. $\mathcal{S}$) iff (i, $i$ denote a sequence of values or terms in place of the arguments or free variables)

(val) for all $p$ in $\text{PRED}$ and every model $\mathcal{D}$ of $Ax$ which is a preinterpretation for $P$, if $p(i) \not\in \text{PTR}(P)$ then $\mathcal{D} \models \mathcal{S}^p$

(eval) A specification $\mathcal{S}$ on $L$ with axioms $Ax$ is computationally valid for $P$ iff for all $p(i)$ in $\text{DEN}(P)$: $Ax \models \mathcal{S}^p[i]$

These definitions need some comments. First, it is clear that (val) implies (eval), that is, if a specification is valid, it is computationally valid. This follows from $\text{DEN}(P) \subseteq \text{PTR}(P)$ for any interpretation $\mathcal{D}$. The converse does not hold. Definition 3.15.1 (eval) can be regarded as a generalization of the notion of computational validity to a class of models.

3.15.2. Definition (Inductive specification (axiomatic view)). A specification $\mathcal{S}$ on $L$ with axioms $Ax$ is inductive for the DCP $P$ iff

$$Ax \cup P[\mathcal{S}]$$

(note that Definition 3.1 is $D = P[\mathcal{S}]$ with the notations of Section 3.11).

3.15.3. Theorem (Axiomatic view of Theorem 3.5). A specification $\mathcal{S}$ on $L$ with axioms $Ax$ is valid for $P$ if it is weaker than some inductive specification $\mathcal{S}'$ on $L$, i.e. there exists $\mathcal{S}'$ such that

(1) $Ax \models P[\mathcal{S}']$ ($\mathcal{S}'$ inductive)
(2) $Ax \models \mathcal{S}' \Rightarrow \mathcal{S}$ (for all $p$ in $\text{PRED}$: $Ax \models \mathcal{S}^p \Rightarrow \mathcal{S}^p$)

Proof. Given $\mathcal{D}$, a model of $Ax$, by hypothesis (1) $\mathcal{D} \models P[\mathcal{S}']$, then $\mathcal{S}'$ is inductive, hence valid (Definition 2.8), i.e. $\forall p(i) \in \text{PTR}(P)$ then $\mathcal{D} \models \mathcal{S}^p[i]$.

Unfortunately, the converse does not hold (demonstration in the appendix).

Now it happens that the completion of a DCP is often regarded as a “specification” of its actual semantics which can be used to prove the validity of a specification (hence its computational validity). The proof method is stated as follows.

3.15.4. Proposition (Proof method with the completion). A specification $\mathcal{S}$ on $L$ with axioms $Ax$ is valid for $P$ if for all $p$ in $\text{PRED}$;

$$\text{IFF}(P) \cup Ax \models p(\bar{x}) \Rightarrow \mathcal{S}^p[\bar{x}]$$

Proof. Obvious as, for any preinterpretation $\mathcal{D}$, model of $Ax$, $\text{PTR}(P)$ satisfies $T_{P, 0}(\text{PTR}(P)) = \text{PTR}(P)$ (see [44]) and is a model of $\text{IFF}(P)$. ☐
However, the converse does not hold (incompleteness of the method). Both methods (Theorem 3.15.3 and Proposition 3.15.4) are incomparable (see demonstration in the appendix), but the inductive one is obviously more modular and thus more tractable. However, both can be combined as will be illustrated in the next section.

3.16. Proving properties of the denotation

The inductive proof method can be used to prove properties of the least term models of a DCP \( P, \text{DEN}(P) \). In fact if \( \mathcal{D} \) is the term interpretation defined by \( \text{DEN}(P) \) then Definition 2.8 becomes (validity and computational validity coincide):

1. \( \text{DEN}(P) \models \text{AND, } p \forall (p(\bar{x}) \Rightarrow \mathcal{I}^P[\bar{x}]) \) (\( \bar{x} \) denotes a sequence of distinct variables in place of the arguments of \( p \)) which is a consequence of \( \mathcal{I} \) inductive, i.e.
2. \( \text{DEN}(P) \models P[\mathcal{I}] \).

This property may be used to prove the validity of formulas of the form (1) by “execution” because the validity of a formula \( F \) in a term model of \( P \) can be deduced from the following statements:

Statement 1: If a formula \( F \) is atomic, it is valid iff it belongs to \( \text{DEN}(P) \). Hence, \( \forall(F) (\exists(F)) \) is semidecidable by running the goal \( F \) in which the universally quantified variables have been skolemized – i.e. replaced by new constants – with a complete strategy of resolution.

Statement 2: If a formula \( F \) is an instance of an axiom of \( P \), it is valid (this is decidable).

Statement 3: Equality axioms (1.5) can be used to deduce new formulas. For example, \( \text{DEN}(P) \models t \) is equivalent to \( \text{DEN}(P) \models -\exists x x = t \land p(x) \).

Statement 4: A formula \( F \) of the form \( \text{AND } \forall(p(\bar{x}) \Rightarrow \mathcal{I}^P[\bar{x}]) \), is valid if \( \mathcal{I} \) is inductive in \( \text{DEN}(P) \).

Statement 5: If \( \text{COMP}(P) \models F \) then \( \text{DEN}(P) \models F \), which is obvious as \( \text{DEN}(P) \) is a model of \( \text{COMP}(P) \). This statement may be used when there is no inductive specification (see (7) below).

3.16.1. Example (addition in the standard model of natural integers). The program \( P \) is the following:

\begin{align*}
\text{(c1)} & \quad \text{int(zero)} \leftarrow \\
\text{(c2)} & \quad \text{int(s(X))} \leftarrow \text{int}(X) \\
\text{(c3)} & \quad \text{plus(zero, X, X)} \leftarrow \\
\text{(c4)} & \quad \text{plus(s(X), Y, s(Z))} \leftarrow \text{plus}(X, Y, Z). \\
\end{align*}

One wants to prove that the following statement holds in \( \text{DEN}(P) \) (commutativity of plus)

1. \( \text{int}(X) \land \text{int}(Y) \land \text{int}(Z) \land \text{plus}(X, Y, Z) \Rightarrow \text{plus}(Y, X, Z) \)
First note that

(2) $\text{DEN}(P) \models \text{plus}(X, Y, Z) \Rightarrow \text{int}(X)$

as the specification reduced to $\text{int}(\text{plus1})$ is inductive, i.e.

$\text{DEN}(P) \models \text{int}(\text{zero})$ and $\text{DEN}(P) \models \text{int}(X) \Rightarrow \text{int}(s(X))$ trivially.

Moreover,

(3) $\text{DEN}(P) \models \text{plus}(X, Y, Z) \land \text{int}(Y) \Rightarrow \text{int}(Z)$
as $\mathcal{F}^{\text{plus}}(\text{plus1}, \text{plus2}, \text{plus3}) : \text{int}(\text{plus2}) \Rightarrow \text{int}(\text{plus3})$ is inductive, i.e.

$\text{DEN}(P) \models \text{int}(X) \Rightarrow \text{int}(X)$ (trivially) and

$\text{DEN}(P) \models (\text{int}(Y) \Rightarrow \text{int}(Z)) \Rightarrow (\text{int}(Y) \Rightarrow \text{int}(s(Z)))$
as $\text{int}(Z) \Rightarrow \text{int}(s(Z))$ is a variant of an axiom.

Thus, to proving the proposition (1), using (2) and (3), reduces to proving (4).

(4) $\text{DEN}(P) \models \text{plus}(X, Y, Z) \land \text{int}(Y) \Rightarrow \text{plus}(Y, X, Z)$

Let us prove that $\mathcal{F}^{\text{plus}}(\text{plus1}, \text{plus2}, \text{plus3}) : \text{int}(\text{plus2}) \Rightarrow \text{plus}(\text{plus2}, \text{plus1}, \text{plus3})$ is inductive, i.e.

(5) $\text{DEN}(P) \models \text{int}(X) \Rightarrow \text{plus}(X, \text{zero}, X)$ and

$\text{DEN}(P) \models (\text{int}(Y) \Rightarrow \text{plus}(Y, X, Z)) \Rightarrow (\text{int}(Y) \Rightarrow \text{plus}(Y, s(X), s(Z)))$
or, better,

(6) $\text{DEN}(P) \models \text{plus}(X, Y, Z) \Rightarrow \text{plus}(X, s(Y), s(Z))$ after a renaming of the variables.
The validity of (5) follows from the inductivity of $\mathcal{F}^{\text{int}}(\text{int1}) : \text{plus}(\text{int1}, \text{zero}, \text{int1})$, i.e.

$\text{DEN}(P) \models \text{plus}(\text{zero}, \text{zero}, \text{zero})$ and

$\text{DEN}(P) \models \text{plus}(X, \text{zero}, X) \Rightarrow \text{plus}(s(X), \text{zero}, s(X))$,
both are instances of clauses.

The validity of (6) follows from the inductivity of $\mathcal{F}^{\text{plus}}(\text{plus1}, \text{plus2}, \text{plus3}) : \text{plus}(\text{plus1}, s(\text{plus2}), s(\text{plus3}))$ with the clauses of $\text{plus}$:

$\text{DEN}(P) \models \text{plus}(\text{zero}, s(X), s(X))$ and

$\text{DEN}(P) \models \text{plus}(X, s(Y), s(Z)) \Rightarrow \text{plus}(s(X), s(Y), s(s(Z)))$,
both are instances of clauses.

It is important to remark that in $\text{DEN}(P)$ the formula $\text{plus}(X, \text{zero}, X)$ is not valid (as it does not belong to $\text{DEN}(P)$), but only (5) is valid with any kind of term base.

Note also the validity of

$(\text{plus}(X, Y, Z) \land Y = \text{zero}) \Rightarrow X = Z$
as $\mathcal{F}^{\text{plus}}(\text{plus1}, \text{plus2}, \text{plus3}) : \text{plus2} = \text{zero} \Rightarrow \text{plus1} = \text{plus3}$ is inductive.
Here is another example.

3.16.2. Example (list permutation). \( P \) is the following:

\[
\begin{align*}
(c_1) & \quad \text{perm}([ ], [ ]) \\
(c_2) & \quad \text{perm}([A|L], [B|M]) \leftarrow \text{perm}(N, M), \text{extract}([A|L], B, N) \\
(c_3) & \quad \text{extract}([A|L], A, L) \\
(c_4) & \quad \text{extract}([A|L], B, [A|M]) \leftarrow \text{extract}(L, B, M) \\
(c_5) & \quad \text{list}([ ]) \\
(c_6) & \quad \text{list}([A|L]) \leftarrow \text{list}(L).
\end{align*}
\]

Let us show that in \( \text{DEN}(P) \) all the arguments of "perm" are lists.

\[
\text{DEN}(P) \models \forall ((\text{perm}(X, Y) \Rightarrow (\text{list}(X) \land \text{list}(Y))) \land
\forall ((\text{extract}(X, Y, Z) \land (\text{list}(Z) \Rightarrow \text{list}(X))))
\]

as the specification:

\[
\{ \langle \text{perm} : \text{list}(\text{perm}1) \land \text{list}(\text{perm}2) \rangle,
\langle \text{extract} : \text{list}(\text{extract}3) \Rightarrow \text{list}(\text{extract}1) \rangle \}
\]

is inductive (this has been proved in Example 3.8 with \( \mathbb{L} \) as model, we prove it with \( \text{DEN}(P) \) as model using a rather automatic method):

\[
\begin{align*}
\text{in } c_1 : \quad & \text{DEN}(P) \models \text{list}([ ]) \land \text{list}([ ]) \\
\text{in } c_2 : \quad & \text{DEN}(P) \models (\text{list}(N) \Rightarrow \text{list}([A|L])) \land \text{list}(N) \land \text{list}(M) \\
& \Rightarrow \text{list}([A|L])) \land \text{list}([B|M])
\end{align*}
\]

obvious

\[
\begin{align*}
\text{in } c_3 : \quad & \text{DEN}(P) \models \text{list}(L) \Rightarrow \text{list}([A|L]) \\
\text{in } c_4 : \quad & \text{DEN}(P) \models (\text{list}(M) \Rightarrow \text{list}(L)) \Rightarrow (\text{list}([A|M]) \Rightarrow \text{list}([A|L]))
\end{align*}
\]

This can be proved using the valid property:

\[
(7) \quad \text{DEN}(P) \models \text{list}([A|M]) \Rightarrow \text{list}(M)
\]

but the specification \( \forall A, M \, (\text{list}1 = [A|M] \Rightarrow \text{list}(M)) \) is not inductive.

It is a consequence of the strongest specification \( \mathcal{F}^{\text{list}} \), which is

\[
\exists n, A_1, \ldots, A_n \, \text{list}1 = [A_1, \ldots, A_n].
\]

This example shows some limitations of the partially automatized method using statements 1–4: such statements can be used to prove the validity of a formula as long as subformulas can be expressed in the form of an inductive specification.

The utility of such a proof method depends on the relationship between the domains of interest and \( \text{DEN}(P) \). It is necessary that the domains of interest are described by a term algebra isomorphic to the Herbrand base used to describe
DEN(P). Then, if the term interpretation of the predicates of \( P \) is exactly \( \text{DEN}(P) \), properties valid in \( \text{DEN}(P) \) correspond to properties valid in the considered interpretation. This is related to the idea of correctness and completeness of the program \( P \) w.r.t. some specification defined on some term interpretation.

In particular, the proof of the formula (1) shows that the predicate "plus" is commutative in \( \mathbb{N} \), the standard model of the natural integers (it is not true in \( \mathbb{E} \) (Section 1.4)). It has been assumed that all the integers are represented in the denotation of the program. For example, the same proof of validity of the formula (1), \( \text{int}(X) \land \text{int}(Y) \land \text{int}(Z) \land \text{plus}(X, Y, Z) \Rightarrow \text{plus}(X, Y, Z) \), could have been performed successfully on the following (obviously incomplete) program!

\[
\begin{align*}
(c_1) & \quad \text{int}(\text{zero}) \leftarrow \\
(c_2) & \quad \text{plus}(\text{zero}, X, X) \leftarrow
\end{align*}
\]

### 3.17. Conclusions on the inductive proof method

Let us summarize the results obtained in this section (Fig. 3–5). We recall the definitions with the unique interpretation view (Sections 3.1–3.14):

- (val1): \( \forall p(\bar{x}) \in \text{PTR}_D(P), D \models S^p[\bar{x}] \) (Definition 2.8)
- (val2): \( \forall p(\bar{x}) \in \text{DEN}(P), D \models S^p[\bar{x}] \) (computational validity)
- (proof1): \( \exists S' \) such that \( D \models P[S'] \) and \( D \models S' \Rightarrow S \)

Results are summarized in Fig. 3. (arrow denotes an implication).

If \( D \) is a term model, both validity definitions become equivalent, leading to Fig. 4. The equivalence is conditioned by the existence of a formula specifying \( \text{PTR}_D(P) \), denoted \( S_{P,D} \).

With the axiomatic view (Sections 3.15 and 3.16):

- (val1'): for all \( D \) s.t. \( D \models Ax, \forall p(\bar{x}) \in \text{PTR}_D(P), D \models S^p[\bar{x}] \)
- (val2'): for all \( p(\bar{t}) \) in \( \text{DEN}(P) \), \( Ax \models S^p[\bar{t}] \)
- (proof1'): \( \exists S'' \) such that \( Ax \models P[S''] \) and \( Ax \models S' \Rightarrow S \)
- (proof2): \( Ax \cup \text{COMP}(P) \models S^p[\bar{x}] \Rightarrow S^p[\bar{x}] \)

Results are summarized in Fig. 5. Some more justifications are given in the appendix.

### 4. Proof method with annotations

The practical usability of the proof method of Theorem 3.5 suffers from its theoretical simplicity: the inductive specifications \( S' \) to be found to prove the validity of some given specification \( S \) will need complex formulas \( S'^p \) since we associate only
one for each $p$ in PRED. It is also shown in [18] that $\mathcal{S}'$ may be exponential in the size of the DCP (to show this result one can use the DCPs transformation into attribute grammars as in [27]). 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 verify 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 [54] or its improvements [25].

4.1. Definition (annotations of a DCP). Given a DCP $P$, an annotation is a mapping $\Lambda$ assigning to every $p$ in PRED a finite set of formulas or assertions $\Lambda(p)$ built as in Definition 2.7. It will be assumed that assertions are defined on $(L, D)$. The set $\Lambda(p)$ is partitioned into two subsets $I\Lambda(p)$ (the set of the inherited assertions of $p$) and $S\Lambda(p)$ (the set of the synthesized assertions of $p$).

The specification $\mathcal{S}_\Lambda$ associated with $\Lambda$ is the family of formulas

\[
\{\mathcal{S}_\Lambda^P : \Lambda \models D \Rightarrow \Lambda \models \mathcal{S}(p)\}_{p \in \text{PRED}}
\]

4.2. Definition (validity of an annotation $\Lambda$ for a DCP $P$, computational validity). An annotation $\Lambda$ is valid for a DCP $P$ if, for all $p$ in PRED in every $D$-based proof tree $T$ of root $p(v_1, \ldots, v_n)$ if $D \models \Lambda(p)(v_1, \ldots, v_n)$ ($n = \rho(p)$) then every label $q(u_1, \ldots, u_m)$ ($m = \rho(q)$) in the proof tree $T$ satisfies: $D \models \Lambda(q)[u_1, \ldots, u_m]$.

In other words, an annotation is valid for $P$ if in every $D$-based 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 also.

Similarly, one may define a notion of computational validity of an annotation for a program $P$ by restricting the hypotheses to the elements of $\text{DEN}(P)$.

An annotation $\Lambda$ is computationaly valid for a DCP $P$ if for all proof trees $t$ in $PT(P)$ of root $p(t_1, \ldots, t_n)$ and all assignments $v$ of the variables of $t$ in $D$ if $(D, v) \models \Lambda(p)(t_1, \ldots, t_n)$ then every label $q(u_1, \ldots, u_m)$ in $t$ satisfies $(D, v) \models \Lambda(q)[u_1, \ldots, u_m]$. 
Note that due to the quantification in the assignments, both definitions do not coincide in the case where D is T. Nevertheless, they coincide for term domains if the annotation is purely synthesized (Definition 4.7) or if the term domain is ground (the assignments are empty). However, by Proposition 4.3, the first definition is stronger than the second one. As noted in the presentation of the inductive proof method (Definition 2.8), only the second definition is of practical interest, but the completeness of the method is achieved with the first definition only.

4.3. Proposition (relationships, validity of \( \mathcal{S}_\Delta \)). If an annotation \( \Delta \) for the DCP P is valid for P, then it is computationally valid for P, but the converse does not hold.

If an annotation \( \Delta \) for the DCP P is valid for P, then \( \mathcal{S}_\Delta \) is valid for P.

If an annotation \( \Delta \) for the DCP P is computationally valid for P, then \( \mathcal{S}_\Delta \) is computationally valid for P.

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

Note that \( \mathcal{S}_\Delta \) can be valid but not inductive (see Example 4.14, second part).

We shall give sufficient conditions ensuring the validity of an annotation and reformulate the proof method with annotations. This formulation is slightly different from that given in [18]. The introduction of the proof tree grammar is a way of providing a syntactic formulation of the organization of the proof.

4.4. Definition (proof tree grammar \((G_{P})\)). Given a DCP \( P = \langle \text{PRED}, \text{FUNC}, \text{CLAUS} \rangle \), we denote by \( G_{P} \) the proof tree grammar of P, the abstract context-free grammar \( \langle \text{PRED}, \text{RULE} \rangle \) such that RULE is in bijection with CLAUS and \( r \) of RULE has profile \( \langle p_1, p_2, \ldots, p_m, p_0 \rangle \) iff the corresponding clause in CLAUS is \( c : p_0(...) \leftarrow p_1(...) \ldots, p_m(...) \).

Clearly, a (syntax) tree in \( G_{P} \) can be associated with every proof tree of \( P \). But not every tree in \( G_{P} \) corresponds to a proof tree of \( P \).

4.5. Definition (logical dependency scheme for \( \Delta \) \((\text{LDS}_\Delta)\)). Given a DCP \( P = \langle \text{PRED}, \text{FUNC}, \text{CLAUS} \rangle \), we denote by \( G_{P} \) the proof tree grammar of P, the abstract context-free grammar \( \langle \text{PRED}, \text{RULE} \rangle \) such that RULE is in bijection with CLAUS and \( r \) of RULE has profile \( \langle p_1, p_2, \ldots, p_m, p_0 \rangle \) iff the corresponding clause in CLAUS is \( c : p_0(...) \leftarrow p_1(...) \ldots, p_m(...) \).

We denote, for every rule \( r \) in RULE of profile \( \langle p_1, p_2, \ldots, p_m, p_0 \rangle \), \( W_{\text{hyp}}(r) \) (resp. \( W_{\text{con}}(r) \)) the sets of the hypothetical (resp. conclusive) assertions which are:

\[
W_{\text{hyp}}(r) = \{ \varphi_k : |k| = 0, \ varphi \in I \Delta(p_0) \text{ or } k > 0, \ varphi \in S \Delta(p_k) \}
\]

\[
W_{\text{con}}(r) = \{ \varphi_k : |k| = 0, \ varphi \in S \Delta(p_0) \text{ or } k > 0, \ varphi \in I \Delta(p_k) \}
\]

where \( \varphi_k \) is \( \varphi \) in which the free variables \( \text{free}(\varphi) = \{ p_1, \ldots, p_n \} \) have been renamed as \( \text{free}(\varphi_k) = \{ p_{k_1}, \ldots, p_{kn} \} \).
The renaming of the free variables is necessary in order to take into account the different instances of the same predicate (if $p_i = p_j = pr$ in a clause for some different $i$ and $j$) and thus different instances of the same formula associated with $pr$.

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

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, [18]. We denote by $\text{hyp}(\varphi)$ the set of all formulas $\psi$ such that $(\psi, \varphi) \in D(r)$ and by $\text{assoc}(\varphi) = p(t_1, \ldots, t_n)$ the atom to which the formula is associated by $\Delta$ in the clause $c$ corresponding to the rule $r$.

4.6. Example (annotation for Example 2.2 and specification $S_2$ (Example 2.9))

$$\text{IA}(\text{plus}) = \text{IA}(\text{plus}) \cup \text{SA}(\text{plus})$$

$$\text{IA}(\text{plus}) = \{ \varphi: \text{ground}(\text{plus}3) \}$$

$$\text{SA}(\text{plus}) = \{ \psi: \text{ground}(\text{plus}1), \delta: \text{ground}(\text{plus}2) \}$$

$$\varphi_0 = \text{ground}(\text{plus}03)$$

$$\varphi_1 = \text{ground}(\text{plus}13)$$

Note that in $r_2$, for example,

$$\varphi_0 = \text{ground}(\text{plus}03)$$

$$\varphi_1 = \text{ground}(\text{plus}13)$$

![Diagram](image-url)
In order to simplify the presentation of $D$, we will use schemes as in [18] representing the rules in $\text{RULE}$ and the LDS of $A$. Elements of $W_{\text{cone}}$ 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 \ldots to $n$ following the left-to-right order for the sons.

4.7. Definition (purely synthesized LDS, well-formed LDS). A LDS for $A$ is purely synthesized iff $\mathcal{A} = \emptyset$, i.e. there is no inherited assertion.

A LDS for $P$ is well formed iff in every tree $t$ of $G_P$ 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 $G_P$ is built with instances of rules $r$ in $\text{RULE}$, in which the local dependency relation $D(r)$ defines dependencies between instances of the formulas attached to the instances of the nonterminals 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 [18]. We recall here only some important results (see [25] for a survey on this question).

4.8. Proposition (known properties of the LDSs).

\begin{itemize}
  \item The well-formedness property of an LDS is decidable.
  \item The well-formedness test is intrinsically exponential.
  \item Some nontrivial subclasses of LDS can be decided in polynomial time.
  \item A purely synthesized LDS is (trivially) well formed.
\end{itemize}

4.9. Definition (Soundness of a LDS for $A$). A LDS for $A \langle G_P, A, D \rangle$ is sound iff for every $r$ in $\text{RULE}$ and every $\varphi$ in $W_{\text{cone}}(r)$ with $\text{assoc}(\varphi) = q(u_1, \ldots, u_m)$ the following holds:

\[ D \vdash \text{AND} \{ \psi[t_1, \ldots, t_n] \ | \ \psi \in \text{hyp}(\varphi) \text{ and } \text{assoc}(\psi) = p(t_1, \ldots, t_n) \} \Rightarrow \varphi[u_1, \ldots, u_m] \]

(Note that the variables $q_i(p_i)$ in a formula $\varphi(\psi)$ are replaced by the corresponding terms $u_i(t_i)$).

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

\begin{itemize}
  \item in $r_1$: $\text{ground}(X) \Rightarrow \text{ground}(X)$
  \hspace{1em} $\text{ground}(\text{zero})$
  \item in $r_2$: $\text{ground}(X) \Rightarrow \text{ground}(s(X))$
  \hspace{1em} $\text{ground}(Y) \Rightarrow \text{ground}(Y)$
  \hspace{1em} $\text{ground}(s(Z)) \Rightarrow \text{ground}(Z)$
\end{itemize}
4.11. **Theorem** (validity of an annotation). Given an annotation $\Delta$ for a DCP $P$, $\Delta$ is valid for $P$ if there exists a sound and well-formed LDS for $\Delta$ for $P$.

**Proof.** We follow the sketch given in [19] showing that all the instances of the formulas $\varphi$ inside the proof tree are valid. Given a proof tree $t$, the relation $D(t)$ is a partial order on the instances of the assertions associated with the nodes of $t$. The proof is done by induction on this order. The minimal elements of $D(t)$ are twofold: inherited formulas of the root (in $IA$) or formulas $\varphi$ associated with a node $n$ with no antecedent following the local dependency scheme corresponding to the context rule $c_1$ if $\varphi$ is inherited or the root subtree rule $c_2$ if $\varphi$ is synthesized (see Fig. 7). By hypothesis (sound $\Delta$), these latter formulas $\varphi$ belong to $W_{\text{conc}}(c) - c$ being the rule with $\text{assoc}(\varphi) = q(u_1, \ldots, u_m)$ and $D \models \varphi[u_1, \ldots, u_m]$.

By the definition of validity, one assumes that the instances of the formulas at the root of $t$ are also valid by $D$.

Now consider any instance of a formula $\varphi$ in $D(t)$ which is not minimal. Taking the formulas which depend on $\varphi$, they correspond to an instance of a local dependency scheme which is from the context rule $c_1$ if $\varphi$ is inherited or in the root subtree rule $c_2$ if $\varphi$ is synthesized (see Fig. 7). By the soundness hypothesis of LDS, the formula $\varphi[t_1, \ldots, t_n]$ is valid in $D$, because a proof tree is built with clause instances and all the formulas $\psi$ it depends on belong to the same rule. $\square$

4.12. **Theorem** (soundness and completeness of the annotation method for proving the validity of specifications) (we use the notations of Definition 3.3 and Theorem 3.5). A specification $\mathcal{S}$ on $(L, D)$ is valid iff it is weaker than the specification $\mathcal{S}_{\Delta}$ of an annotation $\Delta$ with a sound and well-formed LDS, i.e.

(1) there exists a sound and well-formed LDS $\Delta$.

(2) $D \models \mathcal{S}_{\Delta} \Rightarrow \mathcal{S}$.

**Proof.** Soundness follows from Theorem 4.11. Completeness follows from the fact that $\mathcal{S}_{\mathcal{P}, \mathcal{D}}$ is a purely synthesized (thus, well-formed) sound annotation.
4.13. Theorem (soundness and completeness of the annotation method). An annotation $A$ is valid for $P$ iff there exists an extended annotation $A'$ (i.e. $A'$ includes all the formulas of $A$) such that

1. there exists a sound and well-formed LDS$_A$ for $A'$ for $P$,
2. the conjunction of formulas in $I A$ (SA) is stronger (weaker) than the conjunction of formulas in $I A'$ (SA'); or, in short; $\mathbb{D} \models I A \Rightarrow I A'$ and $\mathbb{D} \models S A' \Rightarrow S A$.

Proof. Soundness follows from Theorem 4.11 and Definition 4.2. (for completeness consider $A'$ formed with all the assertions of $A$ plus the synthesized assertion corresponding to the strongest specification $S P_{P,D}$. By hypothesis, in all the $\mathbb{D}$-based proof trees in $P T_D(P)$ of root $p(v_1, \ldots, v_n)$ if $I A(p) [v_1, \ldots, v_n]$ holds then $I A(q) [u_1, \ldots, u_m]$ and $S A(q) [u_1, \ldots, u_m]$ hold everywhere inside the proof tree. However, by definition of $S P_{P,D}$ there is also $\mathbb{D} \models S P_{P,D} [v_1, \ldots, v_n]$ iff $p(v_1, \ldots, v_n)$ is a proof tree root in $P T_D(P)$. Thus, let us consider the clause corresponding to the rule used at the root of the proof tree. All the proof trees in $P T_D(P)$ which use this rule at the root are obtained by $\mathbb{D}$-based instances of clauses which satisfy: (let us denote the clause $P_{o}(t_0) \ldots P_{o}(t_m)$).

\[
\mathbb{D} \models \left( I A(p_0)[t_0] \land_{1 \leq i \leq n} S P_{P,D} [t_i] \right) \Rightarrow S A(p_0)[t_0]
\]

and for all $i, 1 \leq i \leq n$:

\[
\mathbb{D} \models \left( I A(p_0)[t_0] \land_{1 \leq i \leq n} S P_{P,D} [t_i] \right) \Rightarrow I A(p_i)[t_i]
\]

By definition of $S P_{P,D}$ the following holds also (inductive assertion):

\[
\mathbb{D} \models \land_{1 \leq i \leq n} S P_{P,D} [t_i] \Rightarrow S P_{P,D} [t_0]
\]

The corresponding LDS$_A$ is sound, by definition. Furthermore, it is noncircular. In fact, the assertions $S P_{P,D}$ are proved in a purely synthesized manner, then the assertions “$I A(p)$” can be proved and finally the assertions “$S A(p)$”. There is no possibility of cycle as the assertions can be totally ordered (see remark in Section 5.6). To achieve the proof it is sufficient to observe that $I A = I A'$ and $S A' = S A \cup S P_{P,D}$. \(\square\)

We complete this presentation by giving some examples.

4.14. Example (Example 4.10 continued). The LDS is sound and well formed, thus $S P^{\text{plus}} = S P^{\text{plus}}_2$ is a valid specification.

4.15. Example (a valid specification which is not inductive). This example is borrowed from [55] and presented here in a logic programming style: it computes multiples of 4.

\[
\begin{align*}
\text{cl} & : \text{fourmultiple}(K) \leftarrow \text{p}(\text{zero}, H, H, K). \\
\text{c2} & : \text{p}(F, F, H, H) \leftarrow
\end{align*}
\]
c3: \( p(F, sG, H, sK) \leftarrow p(sF, G, sH, K) \)

\[ \mathcal{S}_{\text{fourmultiple}}: \exists N, N \geq 0 \land \text{Fourmultiple} \Rightarrow 4 \times N \]

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

The following annotation \( A \) is considered in [xx]:

\[ \text{IA (four-multiple)} = 8, \text{SA (fourmultiple)} = \{ \text{Formltip'e} \} = \{ S \} \]

\[ \text{LA (p)} = \{ \beta \}, \quad \text{SA (p)} = \{ x, \gamma \} \]

\( x: \exists N, N \geq 0 \land p_2 = p_1 + 2 \times N \)

\( \beta: p_3 = p_2 + 2 \times p_1 \)

\( \gamma: p_4 = 2 \times p_2 + p_1 \)

The assertions can easily be understood if we observe that such a program describes the construction of a “path” of length \( 4 \times N \) and that \( p_1, p_2, p_3 \) and \( p_4 \) are lengths at different steps of the path as shown in Fig. 8. The LDS for \( A \) is shown in Fig. 9. The LDS is sound and well formed. For example, it is easy to verify that the following fact holds in \( \mathbb{D}_1 \):

\( \text{in c}_1: \)

(\( x_1 \land y_1 \Rightarrow \mathcal{S}_{\text{fourmultiple}} \), i.e., \( \exists N, N \geq 0 \land H = \text{zero} + 2 \times N \land K = 2 \times H + \text{zero} \)

\( \Rightarrow \exists N, N \geq 0 \land K = 4 \times N \)

(\( \beta_1 \), i.e., \( H = H + 2 \times \text{zero} \)

\( \text{in c}_2: \)

(\( \beta_0 \Rightarrow \gamma_0 \), i.e., \( H = F + 2 \times F \Rightarrow H = 2 \times F + F \)

(\( \alpha_0 \), i.e., \( \exists N, N \geq 0 \land F = F + 2 \times N \) (with \( N = 0 \)) etc.

Note that as \( \mathcal{S}_A \) is inductive, this kind of proof modularization can be viewed as a way of simplifying the presentation of the proof of \( \mathcal{S}_A \).

![Diagram of the proof](Fig. 8)
Now we consider on the same program a noninductive valid specification $\xi$ defined on $L_2$, $D_2$ (Definition 2.9):

$\xi_{\text{fourmultiple}} : \text{ground}(\text{fourmultiple}_1)$

$\xi^p : [\text{ground}(p_1) \land \text{ground}(p_3)] \Rightarrow [\text{ground}(p_2) \land \text{ground}(p_4)]$

The specification is clearly valid but not inductive since the following does not hold with $D_2$ (term algebra) in $c_1$:

$D_2 \not\models \xi^p \Rightarrow \xi^0_{\text{fourmultiple}}$

i.e.

$D_2 \not\models [(\text{ground}(\text{zero}) \land \text{ground}(H))]$

$\Rightarrow (\text{ground}(H) \land \text{ground}(K))] \Rightarrow \text{ground}(K)$

But it is easy to show that the following LDS (Fig. 10) is sound and well formed:

$I\Delta(\text{fourmultiple}) = \emptyset, \quad S\Delta(\text{fourmultiple}) = \{ \xi_{\text{fourmultiple}} \}$
Dotted lines imply that the LDS is well formed (without circularities). The proofs are trivial.

Note that the corresponding inductive specification is \((x \Rightarrow \beta) \land (\gamma \Rightarrow \delta)\). It is shown in [18] how the inductive specification can be inferred from LDS4.

Note 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 [26] and experimentally studied in [32] (the class of simple logic programs). As shown in [26], this leads to an algorithm of automatic (ground) modes computation for simple logic programs which can compute (ground) modes which are not inductive.

\[ I_A(p) = \{x, \gamma\}, \quad S_A(p) = \{\beta, \delta\} \]

\(x:\ ground(p_1)\)

\(\beta:\ ground(p_2)\)

\(\gamma:\ ground(p_3)\)

\(\delta:\ ground(p_4)\)
4.16. **Power of the method of annotations**

It has been shown in [18] that the annotation method does not permit one to prove more valid specifications than the inductive method does; hence it has the same theoretical power. This is achieved by showing that with a valid annotation with inherited and synthesized assertions it is possible to associate a purely synthesized one (hence inductive) built with the annotation. Furthermore, it is shown that this inductive specification may have an exponential size (with regards to the size of the annotation). This shows that on the one hand the practical complexity of the proof may be improved by using annotations, and on the other, from a practical point of view, it could be much more convenient to use many directional assertions rather than one inductive assertion only.

In fact, even for relatively simple specification like: \( \mathcal{S}_1 \Rightarrow \mathcal{S}_2 \), it is quite natural to consider one inherited (\( \mathcal{S}_1 \)) and one synthesized (\( \mathcal{S}_2 \)). For example, assuming some recursive axiom at nodes 0 and 1, in proving a formula like \( (\mathcal{S}_1(1) \Rightarrow \mathcal{S}_2(1)) \Rightarrow (\mathcal{S}_1(0) \Rightarrow \mathcal{S}_2(0)) \), which may be troublesome to handle, one just has to prove two formulas like \( \mathcal{S}_1(0) \Rightarrow \mathcal{S}_1(1) \) and \( \mathcal{S}_2(1) \Rightarrow \mathcal{S}_2(0) \), for example. Furthermore, the use of inductive assertion becomes totally unnatural if properties which depend on particular upper context inside the tree are considered. The example of Section 5 uses such kind of property.

4.17. **An axiomatic view of the annotation method**

As in Section 3.15, we adapt the definitions as follows.

4.17.1. **Definition (valid annotation (axiomatic view)).** An annotation \( \Delta \) is valid for a DCP \( P \) iff for all \( p \) in \( \text{PRED} \) and every model \( D \) of the axioms \( \text{Ax} \) in every \( D \)-based proof tree \( t \) of root \( p(\overline{v}) \), if \( D \models \Delta(p) [\overline{v}] \) then every label \( q(\overline{u}) \) in \( t \) satisfies \( D \models \Delta(q) [\overline{u}] \).

More interesting from a practical point of view is the notion of computational validity.

An annotation \( \Delta \) is **computationally valid** for a DCP \( P \) (axiomatic view) iff for all proof trees \( t \) of \( \text{PT}(P) \) of root \( p[\overline{r}] \) the following holds (shortened notation):

\[
\text{Ax} \models \Delta(p)[\overline{r}] \Rightarrow \forall q(\overline{u}) \in t \quad \Delta(q)[\overline{u}]
\]

Clearly, if an annotation is valid then it is computationally valid.

The notion of sound LDS is adapted straightforwardly, as expressed in the following theorem (new version of Theorem 4.11).

4.17.2. **Theorem (soundness of the annotation method) (axiomatic view).** An annotation \( \Delta \) is valid for a DCP \( P \) if there exists a LDS\( \Delta \) well formed such that for every \( r \) in \( \text{RULE} \) and every \( \varphi \) in \( \mathcal{W}(r) \) with \( \text{assoc}(\varphi) = q(u_1, \ldots, u_m) \):

\[
\text{Ax} \models \forall \Delta \exists \mathcal{D} \{ \Psi(t_1, \ldots, t_n) \mid \Psi \in \text{hyp}(\varphi) \text{ and } \text{assoc}(\Psi) \models (t_1, \ldots, t_n) \} \\
\Rightarrow \varphi[\overline{u_1}, \ldots, \overline{u_m}]
\]
The proof of Theorem 4.11 with $\mathcal{Ax}$ in state of $\mathbb{D}$ is exactly the same. However, as the completeness of the method (Theorem 3.15.3) does not hold any more with the axiomatic view for purely synthesized LDS, the annotation method is not complete either, even if $\mathcal{Ax}$ is a complete axiom system.

Theorems 4.12 and 4.13 can easily be adapted to the axiomatic view but their "only if" part (completeness) does not hold any more as there is no guarantee of finding a unique set of formulas $\mathcal{S}_{p,0}$ (see the appendix for more details).

5. Illustration of the proof method by annotations

One may think that the proof method by annotation is of limited interest since it has the same power as the fixpoint method in the case of root specifications and the exponential complexity of the inductive assertions rarely occurs. In fact, it is difficult to be convinced of the good value of the method if we base ourselves only on small examples. In turn, if we consider larger and more realistic examples, its superiority appears very naturally. We illustrate this claim by giving an example of a piece of a compiler for a toy language.

This example illustrates also the versatility of logic programming by the way the symbol table is handled in order to maintain partially known references. The example is developed step by step after we have given some informal description of the data structures used (specifications and their interpretation). First, we introduce a specification of the compiler by the definition of the source and object languages, and the mapping defining the translation. Second, we give the data structures and the basic elements needed to understand the annotations which will be used. Then we develop step by step the program by adding new arguments and new annotations in the following order: analysed sentence, symbol table, addresses and generated code. Using the annotation proof method, the compiler will be formally (even if some part of the interpretation is not completely described) proved to be partially correct. The complete program (see P6, Section 5.6) corresponds to the logic program resulting by the transformation described in Section 5.1 from a definite clause grammar. Hence the program is structured by the initial context-free grammar and, additionally, there are auxiliary predicates used to describe manipulations of the symbol table. The proof of partial correctness of this additional program will not be given here as its specification is inductive and it is not difficult to do it. However, the whole proof assumes that the free variables have a defined type. Moreover, for some of them some properties are assumed. For example, it will be assumed that the items of a symbol table are without repetition. Such properties have to be proved on the global program. This will be done separately (Section 5.7).

The language is defined by the following context-free grammar:

\[
\begin{align*}
\text{G:} & \\
\text{prog} & \rightarrow \text{li} \\
\text{li} & \rightarrow \text{lili}
\end{align*}
\]

li stands for "list of instructions"
li → lins
lins → lab : ins
lins → ins
ins → ins
ins → if expr thengoto lab

lins stands for "(labeled) instruction"
lab stands for "label"
conditional jump

Underlined symbols and "::" are terminal; lab, ins and expr are supposed to be generic; they are not described explicitly, in order to keep this example simple. The language generated by G is denoted $L(G)$. Note that the grammar is ambiguous.

Here is a possible program:

$$\text{Prog0: } a : \text{ins if expr thengoto d if expr thengoto a d}$$

The purpose of the compiler is to produce the object code consisting of a list of pairs or triples of the form

$$[\text{number, ins}] \text{ or } [\text{number, expr, number}]$$

in the same order imposed by the sequentializing operator "::" in the source language and such that “number” is an instruction address in the generated code and that all the addresses 1, 2, 3, ... corresponding to the first “number” follow a consecutive order starting from 1.

We will not give a more formal definition as it seems clear enough that to any source program generated by G, say $P$, there corresponds a unique list of object instructions denoted $\text{trans}(P)$, assuming that all the references are solved. For example,

$$\text{trans(Prog0)} = [[1, \text{ins}], [2, \text{expr, 4}], [3, \text{expr, 1}], [4, \text{ins}]]$$

The annotations we will use in the proofs are defined on the interpretation $\mathcal{I}$ (Section 1.4) enriched with some sorts and functions. We give here some of them:

- $\text{trans:} \langle \text{lists, lists} \rangle$ has domain $L(G)$ (lists representing the source programs). Its result is a list which is the generated code.
- $\text{l-table:} \langle \text{lists, lists} \rangle$ its results is a list representing a label table. A source program contains labels which can be stored in the label table without repetition associated with the address of the instruction where they are defined. It will be assumed that a label is uniquely defined. For example, to the source program

$$a : \text{ins if expr thengoto d if expr thengoto a d}$$

corresponds the label table

$$[[a, 1], [d, X]]$$

as this program contains two labels in which only one is defined. However, the program

$$a : \text{ins a : ins}$$
will not have any translation as it is assumed in trans that different instructions have
different address (error handling is not considered here).

The “updating” of the label tables will be defined by a logic program for which we
just give the corresponding inductive assertion assuming that it is well typed. These
assertions can be understood easily if one assumes that a label table is a list of items of
the form [label, integer] in which the integer represents the address of the instruction
in the source program in which it is defined (it may be a variable also).

The label table will be updated by the relation test-or-incl, which adds the item to
the table if it is not already in the table. It is defined here:

\[
\begin{align*}
test-or-incl(T, I, T) & \leftarrow \text{is-already-in}(T, I). \\
test-or-incl(T, I, [I|T]) & \leftarrow \text{is-not-already-in}(T, I).
\end{align*}
\]

\[
\begin{align*}
is-already-in([I|T], I) & \leftarrow \\
is-already-in([I_1|T], I_2) & \leftarrow \text{is-already-in}(T, I_2).
\end{align*}
\]

\[
\begin{align*}
is-not-already-in([I|T], I) & \leftarrow \\
is-not-already-in([I_1|T], I_2) & \leftarrow \text{diff}(I_1, I_2), \text{is-not-already-in}(T, I_2).
\end{align*}
\]

It is not difficult to prove its correctness w.r.t. the following specification (which is
inductive but not the strongest one):

\[
\begin{align*}
\{ & \mathcal{G}^{\text{test-or-incl}}, \mathcal{G}^{\text{is-already-in}}, \mathcal{G}^{\text{is-not-already-in}}, \mathcal{G}^{\text{diff}}, \mathcal{G}^{\text{is}} \} \\
\mathcal{G}^{\text{is}}(i_1, i_2): & i_1 =_N i_2 \quad \text{(this specification will be used later).} \\
\mathcal{G}^{\text{diff}}(\text{diff}1, \text{diff}2): & \text{diff}1 \neq \text{diff}2 \\
\mathcal{G}^{\text{diff}}(\text{diff}1, \text{diff}2): & \text{car}(\text{diff}1) \neq \text{car}(\text{diff}2) \\
\mathcal{G}^{\text{is-already-in}}(i_1, i_2): & 3I, I \equiv i_1 \land I =_{\text{item}} i_2 \\
\mathcal{G}^{\text{is-not-already-in}}(i_1, i_2): & \text{not}(3I, I \equiv i_1 \land \text{car}(I) = \text{car}(i_2)) \\
\mathcal{G}^{\text{test-or-incl}}(\text{to}1, \text{to}2, \text{to}3): & \forall I, I \equiv \text{to}3 \iff (I \equiv \text{to}1 \lor I =_{\text{item}} \text{to}2)) \\
\wedge & \text{noduplic}(\text{to}1) \Rightarrow \text{noduplic}(\text{to}3)
\end{align*}
\]

Some remarks have to be made on these specifications. First, “diff” is not specified.
One may assume here, even if it is not realistic, that it is given by a set of facts diff(a, b)
in which a and b are any possible different labels appearing in a source program. “is”
will not be defined either. It is assumed to be an addition table of the form: a is \(b + c\)
where a, b and c are natural integers. Second, the specifications are expressed in
a many-sorted language. As the program contains not explicitly typed variables, it is
necessary to complete the proof of correctness by showing that the program is well
typed. For example, one will assume that test-or-incl satisfies also the specification: “if
\text{to}1 is a table and \text{to}2 a label then \text{to}3 is a table”. This will guarantee in particular
that when the specification \(\text{diff}1 \neq \text{diff}2\) is assumed, \text{diff}1 and \text{diff}2 are labels, or that
\(I =_{\text{item}} i_2\) in \(\mathcal{G}^{\text{is-already-in}}\) holds on items, i.e. pairs of “label” and “number” which are
tested respectively equal. This verification of well-typedness is given in the Section 5.7 for the whole program P6 (Section 5.6) with an annotation.

Now we start the proof with an annotation concerning the syntax-directed compiler.

5.1. P1: Syntactic analysis

The syntactic aspects of the translation consist in applying a well-known transformation to the grammar G to get a nondeterministic descendant analyser written with definite clauses [56]. We recall here the transformation, but based on difference-lists, and prove its partial correctness using the interpretation I (Section 1.4) enriched with some more functions.

Transformation

If the rule has the form

\[
x \rightarrow t_1, t_2, \ldots, t_n,
\]

where the \( t_i \)'s denote terminal symbols, then generate the clause

\[(1) \quad x([t_1, \ldots, t_n|L] - L) \leftarrow \]

One assumes that the source program is represented by a difference-list in which each element represents a token of the program (i.e. a terminal in the grammar of the source program). If the rule has the form

\[
x_0 \rightarrow w_0 x_1 w_1 \ldots x_k w_k \ldots x_n w_n,
\]

where the \( x_i \)'s denote nonterminal and the \( w_i \)'s sequence of terminal symbols, then generate the clause

\[(2) \quad x_0([w_0|L_0] - L_0), x_1(L_0 - [w_0|L_1]), \ldots, x_k(L_{k-1} - [w_k|L_k]), \ldots, x_n(L_{n-1} - [w_n|L_n]) \]

where \([w_k|L_k]\), \(0 \leq k \leq n\) stands for \([t_1, \ldots, t_n|L_k]\) if \(w_k = t_1 \ldots t_n\) and \(L_k\) only if \(w_k\) is empty.

Partial correctness

With every nonterminal symbol \( x \) of \( G \) we associate the following specification that we state informally:

\( \mathcal{S}_x = x1 \) is a difference-list which represents a terminal string derived from \( x \).

This can be stated formally using the usual transitive closure of the string derivation relation:

\( \mathcal{S}_x = x \searrow \text{repr}(x1) \) (\(x1\) denotes the first argument of \( x \)).

It is easy to show that \( \mathcal{S}_x = \{ \mathcal{S}_x \} \) for all nonterminals \( x \) of \( G \) is inductive on \( \emptyset \), hence valid.

- It is obvious for (1).
- Figure 11 illustrates the proof for (2).
The extra variables $L_0, L_1, \ldots, L_n$ are used to define the unknown strings $a_1, \ldots, a_n$ and if, by hypothesis $x_i \rightarrow a_i$ then the conclusion $x_0 \rightarrow w_0 a_1 w_1 \ldots a_k w_k \ldots a_n w_n$ holds.

The completeness is not proved here. It results, in particular, from the fact that there are as many clauses than there are rules in the grammar $G$.

Applying this transformation to the grammar $G$ leads to the following program (after simplifications which preserve obviously the partial correctness and the completeness):

P1

\begin{align*}
pro(L) & \leftarrow li(L). \\
li(L_0 - L_2) & \leftarrow li(L_0 - L_1), li(L_1 - L_2). \\
li(L) & \leftarrow lins(L). \\
lins([lab \mid L_0] - L_1) & \leftarrow ins(L_0 - L_1). \\
lins(L) & \leftarrow ins(L). \\
ins([ins \mid L] - L) & \leftarrow ins([if, expr, thngoto, lab \mid L] - L) \\
\end{align*}

From now on, we will introduce new arguments one after another such that the annotations may be introduced progressively. The first refinement consists in a new argument which will be removed at the end. It is needed to make a completely formal proof.

5.2. P2: Adding the beginning of a sentence

In order to deal with labels and to build a label table (to store and update backward and forward references) we want to be able to state something like: “all the defined labels of some part of the source program are in the table”. Thus, we introduce an argument whose purpose is to capture what has been already analysed “before”
a nonterminal \( x \) following the left to right order of the terminal sentences derived from the grammar axiom “prog”. It is the purpose of the new first argument added to all the predicates.

P2

\[
\begin{align*}
\text{prog}(M-L) & \leftarrow \text{li}(M-M, M-L). \\
\text{li}(M-L_0, L_0-L_2) & \leftarrow \text{li}(M-L_0, L_0-L_1), \text{li}(M-L_1, L_1-L_2). \\
\text{li}(M-L) & \leftarrow \text{lins}(M, L). \\
\text{lins}(M-N, [\text{lab}, :, L_0]-L_1) & \leftarrow \text{lins}(M-L_0, L_0-L_1). \\
\text{lins}(M, L) & \leftarrow \text{ins}(M, L). \\
\text{ins}(M, [\text{ins} | L] - L) & \leftarrow \\
\text{ins}(M, [\text{if, expr, thengoto, lab} | L] - L) & \leftarrow
\end{align*}
\]

In doing the proof for this particular program, we will define this new argument on the general transformed programs. Thus we again state the transformation, but with a new argument which represents the sentence derived from the grammatical axiom “before” the current nonterminal. We first add a nonterminal “axiom” to \( G \) and a unique rule 0 as follows:

\[
\begin{align*}
0) \ \text{axiom}(M-L) & \leftarrow x(M-M, M-L) \ (\text{for some } x) \\
1) \ x(M, [t_1, \ldots, t_n] | L] - L) & \leftarrow \\
2) \ x_0(M-N, [w_0 | L_0] - L_1) & \leftarrow x_1(M-L_0, L_0-[w_1 | L_1]), \ldots, \\
x_k(M-L_{k-1}, L_{k-1}-[w_k | L_k]), \ldots, \\
x_n(M-L_n, L_n-[w_n | L_n])
\end{align*}
\]

The first argument plays the role of the “beginning” of the sentences derived from “axiom”. Thus, we want to prove something like “in every proof tree of root “axiom”, at every node \( x \) different from “axiom”, \( \text{axiom} \rightarrow \text{repr}(x1) \rightarrow \text{repr}(x2) \rightarrow \) holds” (\( x1 \) is the first argument of \( x \) and \( x2 \) the second – remember that \( x1 \) is the terminal string derived from \( x \), \( x \) some tail of the derived string).

By the previous results concerning the inductive assertion \( \mathcal{J}_1 = x \rightarrow \text{repr}(x2) \) holds. Thus, it is sufficient to consider an inherited assertion associated with the nonterminal symbols different from “axiom”:

\[
\mathcal{J}_1, x^{-1}: \exists x \ \text{axiom} \rightarrow \text{repr}(x1) \rightarrow \text{repr}(x2) \text{ and}
\exists M, N, L \ x1 = M-N \text{ and } x2 = N-L
\]

We prove it by following the logical dependency scheme given in Fig. 14 (note that \( x \) may be any sequence of terminals and nonterminals).

Axiom rule (Fig. 12):

\[
\mathcal{J}_1 \text{ follows from } \mathcal{J}_1:
\]

\[
x \rightarrow \text{repr}(M-L) \rightarrow \text{axiom} \rightarrow \text{repr}(M-L) \rightarrow \text{repr}(M-M) \rightarrow \text{repr}(M-L) x
\]

\( \mathcal{J}_1 \) is the empty sentence (\( x \) is empty).
Second part of the assertion is obvious.

Terminal rule (Fig. 13):
Nothing to prove, as $\mathcal{S}_1(0)$ is inherited.

General rule (2) (Fig. 14):
For the proof we extend the scheme given for the proof of $\mathcal{S}_1$ in (transformation 2) with $M$ and $x1(0)$ the first argument of the root—it has the form $M - L$ by hypothesis (see Fig. 15). By hypothesis also:

$$\text{axiom } \Rightarrow \text{repr}(M - [w_0 | L_0]) \text{ repr}([w_0 | L_0] - L_x)$$

By hypothesis $x1(0)$, i.e. $M - [w_0 | L_0]$ by the second part of the assertion, corresponds to a derivation of the form

$$\text{axiom } \Rightarrow x1(0) \times x.$$
As

\[ x_0 \rightarrow^* w_0 x_1 w_1 \ldots x_k w_k \ldots x_n w_n \]

by hypothesis, axiom \( \rightarrow^* x_1(0) w_0 a_1 w_1 \ldots a_k w_k \ldots a_n w_n x \) and the \( I^k \)'s \( 1 \leq k \leq n \), hold as \( M - L_{k-1} \) corresponds to the derived sentence from “axiom”, preceding \( x_k \). Note that the forms \( M - N \) for \( X_1 \) and \( N - L \) for \( X_2 \) are preserved.

Note that in fact only assertions \( I^e(j), j < k \) are needed to prove \( I^e(k) \).

Program P2 has been simplified as P1 (variables replace identical difference-lists) in a trivial way. The adaptation of the logical dependency schemes to the case of P2 is straightforward. For example, in the third rule, we get the situation depicted in Fig. 16.

5.3. P3: Adding the length of the source program

As we know, there will be as many instructions in the generated code as there are instructions in the source program. Thus, the address of some instruction can be defined by the length of the derived sentence “before” the instruction (+1), i.e. the address of the first instruction corresponding to the code generated for a nonterminal \( x \) is \( \text{length}(X_1) + 1 \). One could use directly this information inside the program. But we want to avoid the explicit use of the first argument, which will be removed later on,
and, moreover, one wants to use simple operations only (addition in place of length). Hence, we add two arguments to all nonterminals in \{li, lins, ins\} denoted \(A_1\) and \(A_2\) (instead of \(x_3\) and \(x_4\) for mnemonic reasons) whose specifications are:

\[
\begin{align*}
\mathcal{F}_1(x_1, A_1): & \quad A_1 = \text{length}(\text{repr}(x_1)) + 1 \\
\mathcal{F}_2(x_1, x_2, A_2): & \quad A_2 = \text{length}(\text{repr}(x_1)\text{repr}(x_2)) + 1
\end{align*}
\]

for all \(x\) in \{li, lins, ins\}

P3

\[
\begin{align*}
\text{prog}(M - L) \leftarrow \text{li}(M - M, M - L, 1, A_2). \\
\text{li}(M - L_0, L_0 - L_2, A_1, A_2) & \leftarrow \text{li}(M - L_0, L_0 - L_1, A_1, A_3), \text{li}(M - L_1, L_1 - L_2, A_3, A_2). \\
\text{li}(M, L, A_1, A_2) & \leftarrow \text{lins}(M, L, A_1, A_2). \\
\text{lins}(M - N, [\text{lab}]; [L_0] - L_1, A_1, A_2) & \leftarrow \text{lins}(M - L_0, L_0 - L_1, A_1, A_2). \\
\text{lins}(M, L, A_1, A_2) & \leftarrow \text{ins}(M, L, A_1, A_2). \\
\text{ins}(M, [\text{ins}[L] - L, A_1, A_2) & \leftarrow A_2 \text{ is } A_4 + 1. \\
\text{ins}(M, [\text{if, expr, thengo}, \text{lab}][L] - L, A_1, A_2) & \leftarrow A_2 \text{ is } A_4 + 1.
\end{align*}
\]

The annotation is proved sound by the dependency schemes of Fig. 17. Most of them are obvious, following the definitions of the arguments.

In rule 3 one needs \(\mathcal{F}_1\) to know that \(N = [\text{lab}]; [L_0]\), thus in terms of instructions \(M - N\) has the same length as \(M - L_0\). Same for \(\mathcal{F}_2\). Similarly for rules 5 and 6.

From these specifications one can observe that \(A_1\) of \(x\) is the address of the first instruction derived from \(x\), as \(A_2\) is the address of the “next” one after \(x\).

5.4. P4: Adding a label table

The label table will be handled by two arguments added to all the nonterminal symbols but the axiom “prog”. They will be denoted \(T_1\) and \(T_2\). \(T_1\) (\(T_2\)) is inherited (synthesized).

Thus for all \(x\) in \{li, lins, ins\}:

\[
\begin{align*}
\mathcal{G}_3: & \quad T_1 \text{ is the table of the labels corresponding to } \text{repr}(x_1). \\
\mathcal{G}_4: & \quad T_2 \text{ is the table of the labels corresponding to } \text{repr}(x_1)\text{repr}(x_2).
\end{align*}
\]

More formally:

\[
\begin{align*}
\mathcal{G}_3: & \quad T_1 = \text{l-table}(\text{repr}(x_1)) \quad \text{for all } x \text{ in } \{\text{li, lins, ins}\} \\
\mathcal{G}_4: & \quad T_2 = \text{l-table}(\text{repr}(x_1)\text{repr}(x_2)) \quad \text{for all } x \text{ in } \{\text{li, lins, ins}\}
\end{align*}
\]

in which it is assumed that \(\text{l-table}(x)\) is the list of all the pairs \([\text{label}, N \text{ or number}]\) in which all labels are those appearing in \(x\). The second element of the pairs may be unknown (a variable \(N\)).

Using two arguments \(T_1\) and \(T_2\) instead of one only avoids the use of sophisticated functions like the merging of two l-tables. We will use the updating function “test-or-incl”.
The program P4 will be given without the arguments A1 and A2 which can be found in P3.

\[
P4 \\
\text{prog}(M - L) \leftarrow \text{li}(M - M, M - L, [ \ ], T_2).
\]
The proofs are obvious, following the logical dependency schemes of Fig. 18.

5.5. P5: Adding the generated code

In order to avoid the use of concatenation of lists, the generated code will be represented by a difference-list. Thus, we will add one argument (denoted C in place of X, whose specification is:

\[ \mathcal{P}_X^*: \text{repr}(C) = \text{trans}(\text{repr}(x2)) \text{ for all } x \in \{\text{prog, li, lins, ins}\} \]

To simplify the presentation of C we do not repeat in P5 the arguments x1, A1, A2, T1, T2.

P5

\[
\begin{align*}
\text{prog}(L, C) & \leftarrow \text{li}(L, C). \\
\text{li}(L_0 - L_2, C_1 - C_3) & \leftarrow \text{li}(L_0 - L_1, C_1 - C_2), \text{li}(L_1 - L_2, C_2 - C_3). \\
\text{li}(L, C) & \leftarrow \text{lins}(L, C). \\
\text{lins}(\text{lab} : |L_0| - L_1, [A_1], T_3) & \leftarrow \text{test-or-incl}(T_1, [\text{lab}, A_1], T_3), \text{ins}(L_0 - L_1, C). \\
\text{ins}(L, C) & \leftarrow \text{lins}(L, C). \\
\text{ins}([\text{if, expr, thengoto lab} |L| - L_1, [A_1, \text{expr}, X] |C| - C) & \leftarrow A_2 \text{ is } A_1 + 1. \\
\text{ins}([\text{if, expr, thengoto lab} |L| - L_1, [A_1, \text{expr}, X] |C| - C) & \leftarrow A_2 \text{ is } A_1 + 1, \text{test-or-incl}(T_1, [\text{lab}, X], T_2). \\
\end{align*}
\]

The proofs are obvious, following the logical dependency schemes of Fig. 19.

Note that a label in the generated code may remain a variable if it is not defined in the source program and if a label is defined twice (i.e. with different addresses) object code will not be there as there is no proof tree for P4 in this case.

The proof of soundness of the logical dependency schemes has now been completed. As the variables appearing in the first argument of li, lins and ins are not used in the arguments A1, A2, T1, T2 or C, it is possible to remove the first argument for these predicates. Thus, the final program is the following:
Fig. 18. Sound logical dependency scheme for P4.
Fig. 19. Sound logical dependency scheme for P5.
5.6. **P6: Final program**

\[ prog(L, C) \leftarrow li(L, 1, A, 2, [ ], T, 2, C) \]

\[ li(L_0 - L_2, A_1, A_2, T_1, T_2, C_1 - C_3) \leftarrow li(L_0 - L_1, A_1, A_3, T_1, T_3, C_1 - C_2) \]

\[ li(L_1 - L_2, A_3, A_2, T_3, T_2, C_2 - C_3) \]

\[ li(L, A_1, A_2, T, 2, C) \leftarrow lins(L, A_1, A_2, T_1, T, 2, C) \]

\[ lins([lab, : |L_0|] - L_1, A_1, A_2, T_1, T_2, C) \leftarrow \text{test-or-incl}(T_1, [lab, A_1], T_3) \]

\[ lins(L_0 - L_1, A_1, A_2, T_3, T_2, C) \]

\[ \text{test-or-incl}(T_1, [lab, X], T_2) \]

The annotations are: (X is the removed first argument)

**Inherited:** for all x in \{li, lins, ins\}:

\[ x_1 \Rightarrow \exists x \text{ prog} \rightarrow \text{repr}(x_1) \text{ and } \exists M, N, L \text{ x}_1 = M - N \text{ and } x_2 = N - L \]

\[ A_1 = \text{length(repr}(x_1)) + 1 \]

\[ A_2 = \text{l-table(repr}(x_1)) \]

**Synthesized:**

\[ \forall x \Rightarrow x \rightarrow \text{repr}(x_2) \text{ for all } x \in \{\text{prog, li, lins, ins} \} \]

\[ A_2 = \text{length(repr}(x_1) \text{ repr}(x_2)) + 1 \text{ for all } x \in \{\text{li, lins, ins} \} \]

\[ T_2 = \text{l-table(repr}(x_1) \text{ repr}(x_2)) \text{ for all } x \in \{\text{li, lins, ins} \} \]

\[ \text{repr}(C) = \text{trans(repr}(x_2)) \text{ for all } x \in \{\text{prog, li, lins, ins} \} \]

and:

\[ \forall x \Rightarrow \text{test-or-incl}, \text{is-already-in}, \text{is-not-already-in}, \text{diff}, \text{diff}, \text{ins} \]

The logical dependency schemes are obtained by merging all the previous partial dependency schemes:

As the logical dependency scheme is sound and noncircular, the annotation is valid; hence, every proof tree of root prog(S, C) satisfies:

\[ \forall x \Rightarrow \text{prog} \rightarrow \text{repr}(S) \text{ and } \forall x \Rightarrow \text{repr}(C) = \text{trans(repr}(S)) \]

**Remark.** The claim that the LDS is noncircular may not be obvious at first glance. We did not give in this paper any algorithm to solve the problem and we only help the reader to get convinced. However, in most of the cases the following property can be used: if there is a total order between the formulas of \(A(p)\) for each predicate \(p\) such
that there is no cycle provoked by the local dependencies in \( p \) in any rule, then the LDS is noncircular. In this case the LDS is said \( l \)-ordered.

In general the problem whether a LDS is \( l \)-ordered is NP-hard. In practice it is relatively easy to find such an order. We did this in the demonstration of the Theorem (4.13) (using the order: \( \mathcal{D}_k \rightarrow \text{IA}(p) \rightarrow \text{SD}(p) \) and any order for the formulas inside \( \text{IA} \) and \( \text{SD} \)). In this example the order in which the assertions have been introduced progressively defines such a total order. Hence, the result of noncircularity.

5.7. The program \( P6 \) is well-typed

One uses the following annotation which shows that all arguments of the predicates in a complete proof tree of root “prog” respect their type and that symbol tables are...
represented by a list of items whose first elements are different ground atoms and the second is an integer or a variable (all this information is condensed in "list-of-items"). Note that there is no inherited assertion associated with the root "prog". In particular, the "atomic" labels may be represented by any kind of ground term.

**Inherited assertions**

- $\mathcal{I}_4$: list-of-items($x4$) $\land$ noduplic($x4$), $x \in \{li, lins, ins\}$
- $\mathcal{I}_{3}^{test-or-incl}$: list-of-items($toi1$) $\land$ item($toi2$)
- $\mathcal{I}_5$: list-of-items($x1$) $\land$ item($x2$), $x \in \{is-already-in, is-not-already-in\}$
- $\mathcal{I}_6^{diff}$: item($diff1$) $\land$ item($diff2$)
- $\mathcal{I}_6^{diff}$: atom($diff1$) $\land$ atom($diff2$)
- $\mathcal{I}_5$: integer($x2$), $x \in \{li, lins, ins, is\}$

**Synthesized assertions**

- $\mathcal{U}_4^{prog}$: is-a-d_list($x1$), $x \in \{prog, li, lins, ins\}$
- $\mathcal{U}_5^{prog}$($L_C$): is-a-d_list($C$) $\land$ well-typed(repr($C$))
- $\mathcal{U}_5^{prog}$: is-a-d_list($x6$) $\land$ well-typed(repr($C$)), $x \in \{li, lins, ins\}$
- $\mathcal{U}_5^{diff}$: list-of-items($li5$) $\land$ noduplic($li5$), $x \in \{li, lins, ins\}$
- $\mathcal{U}_5^{test-or-incl}$: list-of-items($toi3$)
- $\mathcal{U}_5^{def}$: integer($x3$), $x \in \{li, lins, ins, is\}$

Two assertions ($\mathcal{I}_5^{def}$, $\mathcal{U}_5^{prog}$) need some comments. In $\mathcal{I}_5^{def}$ one requires "well-typed(repr($L$))"; as $L$ is a difference list by assertion $\mathcal{I}_5^{prog}$, the "well-typed" predicate
means only the labels of the source program must be ground atoms (needed to satisfy "list-of-items", "item" and "atom" properties). Furthermore, in $\mathcal{H}_2^{\text{eq}}$ "well-typed (repr(C))" means that the generated code C is a list of pairs or triples whose components have the right type described at the beginning of Section 5 (in particular, the first element is an integer).

The proofs are obvious. We just draw the local dependency scheme (Fig. 21), most of them are obvious. Note that some of them are just inherited.

6. Comparison with other works on partial correctness in logic programming

The first presentation of use of the inductive proof method to establish the computational validity of a specification appears in [12] mainly based on the axiomatic view of Sections 3.15 and 3.16. When induction in the theories defined by what is called here Ax (axioms defining the "input relations" or in some sense the pre-interpretations) is performed, it is called structural induction (on the "input relations") in the sense of Burstall and Darlington [11].

When a formula is proved using statement 4 in (3.16), i.e. using an inductive specification in $\text{DEN}(P)$, it is called computational induction in the sense of Manna [45].

Computational induction, as structural induction, can be used to prove general properties about the program, holding in $\text{DEN}(P)$. Clark remarks also that the "pure" inductive assertion method, called fixpoint induction by Park [47], is a sound rule to prove the computational validity. He calls this rule the "consequence verification method".

In most of his examples Clark considers that the set of axioms Ax is defined by a logic program. Hence statement 5 (Section 3.16) is extensively used (Ax is the completion of a logic program).
Fig. 21.
Fig. 21. Sound and noncircular logical dependency scheme for proving the well-typedness of P6.

We adopted the same point of view in Section (3.16) showing that these different kinds of inductions are the same if one considers that the axioms Ax are also definite clauses.

The first presentation of the proof method by annotations appears in [18] where it is restricted to the use of assertions to prove valid specifications. We have given here a slightly more general presentation of the method (adapted to the case of logic programming): we use the annotation method to prove properties holding inside the proof trees, not only at the root. With some adaptation the method can be used to prove dynamic properties.

For example in [33] a scheme analogous to a L-LDS is used to prove run-time properties, but if ground proof trees are considered, the kind of assertions used can be viewed as assertions holding in the proof trees; hence, in particular at the root.
In [14] the same kind of L-LDS is proposed but with an additional feature: the unification is completely axiomatized and its axioms together with a restricted assertion language on terms domains serve to remove from the formulas to be proved any reference to the unification process. They justify their approach by an analogy with the Hoare’s proof method: given a precondition, the postcondition assumes that the programs halts with success. We did deal with the same idea here: “to halt” means that a complete proof tree has been constructed. The difference with their approach comes from the fact that they take into account the operational semantics, that is to say, the way the proof trees are constructed. This is necessary in the case of study of run-time properties and halting problems. The same kind of analogy with the Hoare’s method is developed in [1], where not only a simple deductive system is proposed to prove run-time properties but also input/output relations. Basically, all these systems (at least for the part concerning the program) are modelling same kind of induction on the program structure. They leave possibility of a greater modularity which become necessary to handle big programs.

In Bossi and Cocco [9] an annotation based on a L-LDS is used to prove partial correctness of “modules” w.r.t. a specification (computational validity). They use the axiomatic view, and we have shown that the restriction to L-LDS is not necessary.

In Hogger [35, 39], it is assumed that the axioms Ax are a DCP which includes another definition of the predicates in PRED. The definition of “inductive for P” becomes just: \( S \models P \), i.e. the clauses of P are theorems in \( Ax = S \). Theorem 3.15.3 is a reformulation of the sufficient criterion given by Hogger, in this particular case, when Ax contains another axiomatisation of P also.

The same idea is used in [41] to prove the equivalence of logic programs. Assuming for simplicity that both programs define (in different ways) the same predicate, two programs \( P \) and \( P' \) are equivalent (called CAS equivalence) iff \( DEN(P) = DEN(P') \). One way to obtain the result is to prove by some method that the denotation of one program is a model of the other and conversely, i.e. \( DEN(P) = P' \) and \( DEN(P') = P \).

In [49] the inductive proof method is used to validate specifications (computational validity) expressed on Herbrand interpretations, but the method itself is not investigated.

Kanamori [42, 49] developed extensively the idea that execution of a DCP can be used to prove that general formulas are logical consequences of \( COMP(P) \) or valid in \( DEN(P) \). The work is restricted to DCPs and to formulas called S-formulas (of the form \( \forall x \exists y F(x, y) \)). In [42] the “extended execution deduction rule”, which is an extension of the SLDNF resolution, is defined and proved to be complete; i.e. given a DCP \( P \) and an S-formula \( F \), \( COMP(P) = F \) if and only if starting with \( F \), the extended execution deduction rules applied to \( P \) permit to derive the “true” goal. This deduction rule is implemented in the system ARGUS [43] with the induction proof method which is used to prove S-formulas valid in \( DEN(P) \). These ideas have given rise to many improvements [36] and analogous ideas have been studied in [40].

In Deville [31] and Deransart and Ferrand [21, 23, 24] the extension of the inductive proof method for logic programs with negations (normal programs) is
presented. For DCPs the completion \( \text{COMP}(P) \) has a unique least term models; for normal programs it is not nomore true and the completion may have many uncom-parable minimal term models. Deville assumes that there exists a unique term model. This property is undecidable in general but is verified by construction of the (normal) program. His method needs to know the strongest specification and is introduced with the purpose of building correct programs rather than proving them correct afterwards. Deransart and Ferrand have basically the same approach (the specification, if expressed in a term model, is a well-founded model of \( \text{COMP}(P) \) – well founded in a sense which is out of the scope of this paper and it is explained in the comprehensive version of [21]), but they give a modular version of the proof method for partial correctness which reduces to the inductive proof method in case of a DCP, and is suitable for weaker specifications. However, the proof of weaker (or partial) specifica-tions needs stronger (or larger) specifications of completeness.

7. Conclusion

In this paper, we have investigated the problem of proving the partial correctness of a logic program – restricted to definite clauses and some possible extensions – w.r.t. a specification in an unified framework. We have considered two cases: “informal” proofs (i.e. performed in some “known” domain) or “formal” proofs (i.e. performed in some axiomatized domains), and we have introduced two proof methods: the inductive method and the annotations method, the first one being a particular case of the second one.

We have introduced two notions of validity with the purpose of obtaining completeness results of the methods: “validity” and “computational validity”. From the point of view of a (logic programming) programmer the second notion is the only interesting one.

Different notions of “completeness” have been used.

(1) Completeness of a set of axioms: Obviously when the axiomatic view is used – and automatized proofs are performed – if the set of axioms is not complete (i.e. if the theory is not decidable) it does not make any sense to speak of other concepts of completeness: any method will remain incomplete.

(2) Completeness of the method (inductive assertions or annotations) expressed by a necessary condition for validity. Neither with the computational validity nor in general, if the axiomatic view is taken, even with a decidable theory, the completeness can be reached. If so, there is still a problem.

(3) Relative completeness. Completeness of the method can be reached only if the specification language is powerful enough. We have shown that if one restricts the specification language to the first-order logic, the completeness of the method does not hold in general, even for pure DCP’s.

The presented proof methods can be viewed as new deduction rules, and we have shown that they can be used for more general purposes than just to prove the validity of specifications: proof of validity of general formulas in one model (DEN) or proof of
properties inside the proof trees. The annotation method, whose adaptation to the field of logic programming is new, seems to be particularly well suited for this latter purpose. However, we have shown that the annotation method cannot increase the power of other known methods.

We have compared this presentation of the proof methods with other known works, and we have shown that all of them use with a slightly different formalism, or different conditions, the same method: the inductive one or fixpoint one or in the best case the annotation method with a L-LDS.

We think that the existence of simple and powerful proof methods of validity is one of the characteristics of logic programming and these make it attractive. We carried out this extensive study because of two main reasons:

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

(2) Valid specifications are the basic elements used in the proofs of all other desirable logic program properties as completeness, “run-time” properties, termination such as shown in [22] or safe use of the negation [44]. For example, any proof of termination with regard to some kind of used goals and some strategy will suppose that, following the given strategy, some subproof 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 independent of any other property.

As most of the other manipulations on logic programs (proof of other properties, program transformations, etc.) use valid specifications, one may expect that the same results of (in)completeness hold also for these methods. This is fortunately not a reason not to try to validate a program and to develop good methods well adapted to the kind of property one wants to verify.

Acknowledgment

We are indebted to B. Courcelle, with whom most of the basic ideas have been developed in the field of attribute grammars, and to G. Ferrand, J.P. Delahaye, S. Renault, N. Francez and an anonymous referee who suggested some questions and helped to clarify previous versions of this text.

Appendix

Missing proofs of Section 3.15.

Claim A.1. The only-if part of the Theorem 3.15.3 does not hold.
Proof. By hypothesis, for every $\mathcal{D}$ model of $\text{Ax}$, if $p(\bar{v}) \in \text{PTR}_{\mathcal{D}}(P)$ then $\mathcal{D} \models \mathcal{S}^p[\bar{v}]$. Let us consider some $\mathcal{D}$ and $\mathcal{S} = \mathcal{S}_{\mathcal{P}, \mathcal{D}}$; $\text{Ax} \models P[\mathcal{S}_{\mathcal{P}, \mathcal{D}}]$ does not hold necessarily in all models of $\text{Ax}$ (it does for $\mathcal{D}$ by hypothesis). In other words there may be different formulas $\mathcal{S}_{\mathcal{P}, \mathcal{D}}$ for different models $\mathcal{D}$, i.e. there is no unique formula which is inductive in all models of $\text{Ax}$ and implies the valid specifications $\mathcal{S}$.

The proof is achieved by a counterexample.

Consider the following program:

$$p(a) \leftarrow q(a, b), q(b, c)$$

$$q(x, x) \leftarrow$$

and the specification

$$\mathcal{P}^p: p1 = c, \quad \mathcal{Q}^q: q1 = q2$$

expressed in the language which contains only three constants, \{a, b, c\}, and one binary predicate \{=\}.

Let $\text{Ax}$: $\{\forall x, x=x\}$ be the only axiom (which holds in all interpretations).

The specification $\mathcal{S}$ is valid. In fact, any $\mathcal{D}$-based proof tree of root $p$ requires that all constants are interpreted as the same value.

But there is no inductive specification $\mathcal{U}$ such that $\text{Ax} \models \mathcal{U}^p \Rightarrow \mathcal{S}^p$.

In fact, with this limited axiom one can prove only formulas which are an instance of this axiom. So the strongest inductive formula is:

$$\mathcal{U}^p: p1 = a, \quad \mathcal{Q}^q = \mathcal{S}^u$$

But $\{\forall x, x=x\} \not\models p1 = a \Rightarrow p1 = c$. □

Claim A.2. The only-if part of the Proposition 3.15.4 does not hold.

Proof. Assume $S$ is a valid specification and that $\mathcal{D}$ is a model of $\text{Ax} \cup \text{IFF}(P)$. Then given a predicate $p$ of $P$, all the elements $p(\bar{v})$ of $\text{PTR}_{\mathcal{D}}(P)$ are in $\mathcal{D}$. However, $\text{PTR}_{\mathcal{D}}(P)$ is the least fixpoint and the converse does not necessarily hold [44]. By hypothesis, the implication $p(\bar{x}) \Rightarrow p[\bar{x}]$ holds for the elements in $\text{PTR}_{\mathcal{D}}(P)$ but not necessarily for all "atoms" in $\mathcal{D}$. The same example is also a counterexample for this method. □

Claim A.3. Both proof methods of Section 3.15 (3.15.3 and 3.15.4) are incomparable.

This is left as a conjecture. We give only some intuition.

Consider both properties:

(1) $\text{Ax} \cup \text{IFF}(P) \models p(\bar{x}) \Rightarrow \mathcal{S}^p[\bar{x}]$ for all $p$ in $\text{PRED}$.

(2) There exists $\mathcal{S}'$ such that

(a) $\text{Ax} \models P[\mathcal{S}']$

(b) $\text{Ax} \models \mathcal{S}' \Rightarrow \mathcal{S}$

We show (1) does not imply (2) and (2) does not imply (1).
(2) does not imply (1). Let \( D \) be a model of \( \text{Ax} \cup \text{IFF}(P) \), then by 2(a) \( D \models P[\mathcal{S}''] \) and by 2(b) \( D \models \mathcal{S}' \Rightarrow \mathcal{S} \). Given an assignment such that \( D \models p(v) \), is it true that \( D \models \mathcal{S} \)?

It is, if one can ensure that \( D \models \mathcal{S}'[v] \). However, \( D \), by hypothesis, corresponds to one of the fixpoints of \( P \). So \( \mathcal{S}' \) does not correspond necessarily to the same fixpoint (it can be a smaller one).

(1) does not imply (2). Assume (1). For every model of \( \text{Ax} \cup \text{IFF}(P) \) assume that there is a specification \( \mathcal{S}' \) characterizing the interpretation of the predicates of \( P \) in \( D \) such that (3) \( D \models P[\mathcal{S}''] \) and \( D \models \mathcal{S}' \Rightarrow \mathcal{S} \). Moreover, every model of \( \text{Ax} \) can be extended into a model \( \text{Ax} \cup \text{IFF}(P) \) which satisfies (3). However, there is no guarantee that the specification \( \mathcal{S}' \) will be the same for all models of \( \text{Ax} \).

In other words, the strongest inductive formula by the second method may define in some model of \( \text{Ax} \) an interpretation which is not a model of \( \text{IFF}(P) \) and strictly includes the greatest fixpoint of \( T^P \), hence 2(b) may not hold.

As we did not yet find a satisfactory counterexample, we leave it as a conjecture.

References


