Concept, A Proof System for Classifying Datalog Predicates

Miguel-Angel Oros* & Paul Y Gloess

LaBRI, ENSERB & Université Bordeaux I, 351 Cours de la Libération, F-33405 Talence Cédex, France.
Fax (33) 56 84 66 69
e-mail: {oros, gloess}@labri.u-bordeaux.fr

Abstract

We propose an exact method based of natural deduction for inferring a partial order among predicates used or defined in a Datalog program. Our approach is compared to other approaches based on abstract interpretation. The method has been implemented in Prolog. Decidability issues are briefly discussed.

Keywords: Datalog, classification, inheritance.

1. Introduction and Motivation

In the context of object-oriented logic programming, e.g., P-term theory [AITK91, AITK86], a signature defines a priori a partial order among sorts. Sorts correspond to classes and partial order to class inheritance, e.g. politician and civilian are a kind of person; P-terms are terms built from sorts and features, using a keyword syntax. Features correspond to attributes in object oriented programming. P-terms have a type semantics: they denote a set of objects, given some interpretation of sorts as sets and features as functions. A typical P-term is

X:person(age⇒Y,boss⇒engineer(father⇒X,weight⇒Y))

and denotes the set of persons X whose age Y is equal to the weight of their boss who is an engineer and is also their father.

P-terms may be used as arguments of predicates. It is tempting to abolish the traditional distinction between predicate symbols and function symbols by calling them sorts and letting P-terms be literals. However, this may yield incoherence when some clause tries to define a sort which is also constrained by the partial order included in the signature. It is safe for literals used in the body of a clause, as in U-Log [GLOE91].

It is however a more and more common view that imposing a partial order among sorts from the outset is too rigid an approach, and that logic itself is perfectly suitable for the expression of a partial order, using logical implication. Thus Backofen and Smolka's logical explanation of feature types [BACK92] does not assume any partial order among sorts; Frühwirth, Shapiro, Vardi and Yardeni advocate in [FRUH91] the use of unary predicates for describing types. Hanus [HANU91] uses a two-level Prolog language, defining parametric sorts using a meta-level, and using them in the concrete level.
Closer to the A. I. community, terminological theories of concepts, e.g. [NEBE87] study how to derive a subsumption partial order among concepts and among roles. In this context, sorts are called concepts, and features are called roles.

Being able to derive a partial order among defined sorts is extremely useful for optimization purposes. \(\Psi\)-term unification use of partial order information saves inference steps, but does not assume this partial order to be given \emph{a priori} rather than automatically derived. Note that following Backofen and Smolka’s work, \(\Psi\)-term nested syntax may be translated into flat constraints, with sorts as unary predicates and features as binary predicates. This translation is most appropriate in our context, since we do not assume a functional interpretation of features, unlike in \(\Psi\)-term or feature type theories.

As already hinted, logical definition of sorts generalizes partial order declarations. This is illustrated by the following example, which will serve as a basis for illustrating various approaches including our approach:

\begin{verbatim}
person(X) :- politician(X).
person(X) :- civilian(X).
influent(X) :- politician(X).
influent(X) :- person(X), friend(X,Y),
influent(Y).
\end{verbatim}

Figure 1. The politician example.

The first two clauses are equivalent to the sort partial order declarations:

\begin{align*}
politician & \leq \text{person} \\
civilian & \leq \text{person}
\end{align*}

The remaining clauses define an influent person as either a politician, or a person who has an influent friend. From there, we would like to derive as a logical consequence the fact that

\[ \text{influent} \leq \text{person} \]

or more formally:

\[ \forall X [\text{influent}(X) \Rightarrow \text{person}(X)] \]

Several methods for inferring types from logic program, e.g., [YARD91, SOLN93] have been conceived. These approaches fall in the general context of abstract interpretation initiated by Cousot [COUS77], and more specifically rely upon an abstraction function \(a\), called the “distributive closure”, which computes upper bounds of subsets of the Herbrand universe of a given program \(P\), by abolishing relations between arguments of predicates.

The \(T_P\) inference operator associated with \(P\) is approximated by the functional composition \(\alpha \circ T_P\), denoted \(T_P^\alpha\). This approximation yields upper bounds of the \(T_P\) least fixed point, with the inclusion relations:

\[ \text{lfp}(T_P) \subseteq a(\text{lfp}(T_P)) \subseteq \text{lfp}(T_P^\alpha) \]

Solnon’s approach, for instance, consists in building equations whose unknown are types, and, instead of solving them, derive inheritance relations among these types. However, the approximation sometimes yields incorrect results. As was pointed to us by Baudouin Le Charlier, these errors seem to stem from the fact that in order to prove \(u \leq v\) one tries to prove \(u' \leq v'\), where \(u'\) and \(v'\) are upper bounds of \(u\) and \(v\) respectively. One should use an upper bound \(u'\) of \(u\), but a lower bound \(v'\) of \(v\), so that \(u \leq v\) is a logical consequence of \(u' \leq v'\).
2. Inheritance as Logical Implication

Our approach does not fall in the context of abstraction interpretation. It uses the completed version of a program P denoted by "comp(P)" [LLOY87] and an inference system based on natural deduction.

We define inheritance in terms of logical implication. Precisely, let B be a basis of a predicate logic with equality, with no function symbol (no constant symbol either), and predicate symbols of arity 1, called sorts, and predicate symbols of arity 2, called features. Variables are denoted X, Y, Z, ...

A logic program on this basis B is thus a Datalog program, which may be assumed normalized (i.e., with all variables distinct in clause heads), since equality is permitted in clause bodies.

Given a logic program P on this basis B, and two sorts a and b, we say that sort a inherits from sort b (with respect to P), written a\lessdot\lessdot b, if and only if

\[ \text{comp}(P) \models \forall X[a(X)\Rightarrow b(X)] \]

Inheritance among undefined sorts can be specified by means of additional clauses. For instance, if person, civilian and politician are undefined sorts, and we wish to specify the partial order

\[
\text{politician} \leq \text{person} \\
\text{civilian} \leq \text{person}
\]

among them, we can do it by adding two clauses that define the person sort:

\[
\text{person}(X) :- \text{politician}(X). \\
\text{person}(X) :- \text{civilian}(X).
\]

Hence it is not necessary to assume any initial partial order among sorts.

Thus, our problem in general consists whether a given sort a inherits from a given sort b, with respect to a logic program P under consideration, that is, whether we have

\[ \text{comp}(P) \models \forall X[a(X)\Rightarrow b(X)] \]

Two generalizations are actually possible: we can let predicates of any arity belong to the basis B; we can also generalize sort inheritance to inheritance among predicates of same arity. Given two predicates p and q of same arity, we say that p inherits from q (with respect to P), if and only if

\[ \text{comp}(P) \models \forall X[p(X)\Rightarrow q(X)] \]

where \( \overline{X} \) denotes a vector of distinct variables, the length of which is the arity of p and q.

We actually need to add axioms to \( \text{comp}(P) \), which specify that undefined concepts of arity 1 have disjoint interpretations: "contradict(P)" is the set of all formulas of the form:

\[ \forall X \neg[a(X)\wedge b(X)] \]

where a and b are distinct predicates of arity 1, used in P but undefined by P. This axiom is the "distinct sort disjointedness" axiom of Backofen and Smolka's recursive theory of feature terms [BACK92].

3. Natural Deduction of Inheritance

Proving that predicate p inherits from predicate q with respect to P amounts to prove a universally quantified logical implication, from \( \text{comp}(P)\wedge\text{contradict}(P) \). We first give the idea of our proof method, which is an
extension of Prolog inference, and then present this method as a formal inference system.

In order to prove the $p(\overline{X}) \Rightarrow q(\overline{X})$ implication, we assume $p(\overline{A})$, where $\overline{A}$ is a vector of distinct constants, and then try to prove $q(\overline{A})$. With a Prolog system, assuming $p(\overline{A})$ may be simulated by asserting this fact. Then, if Prolog can prove $q(\overline{A})$ using its own backward inference, we are done; otherwise, we replace $p(\overline{A})$ with its disjunctive expansion computed by unfolding predicate $p$ using its definition in $\text{comp}(P)$. This yields to a case analysis, with as many cases as conjunctions of atoms in the disjunction. Each case is treated by assuming the corresponding conjunction, and trying to prove $q(\overline{A})$ again using Prolog backward inference.

For the politician example in Figure 1, the completed program is precisely:

$$\forall X \quad \text{person}(X) \Leftrightarrow \text{politician}(X) \lor \text{civilian}(X)$$

$$\forall X \quad \text{influent}(X) \Leftrightarrow \text{politician}(X) \lor \exists Y \ (\text{person}(X) \land \text{friend}(X,Y) \land \text{influent}(Y))$$

Our natural proof of "influent:;;person", as produced by our implemented system, is exactly that of figure below. Note that the first line (which is underlined) is an input to the Prolog system, while the remainder is the corresponding output. The "implies" predicate requires the proof of a universally quantified implication whose hypothesis is the first argument, and conclusion is the second one. The program has been supplied under the form of Figure 1.

We are asked to prove that: for all $X$, influent$(X)$ implies person$(X)$.
Let influent0 be a new constant.
Assume: influent(influent0).
We must prove: person(influent0).
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: influent(influent0).
There are 2 cases to consider:
Case 1:
Assume: politician(influent0).
We must prove: person(influent0).
Let us try Prolog on this goal.
Prolog succeeds!
End of Case 1.
Case 2:
Let friend1 be a new constant.
Assume: person(influent0),
friend(influent0,friend1),
influent(friend1).
We must prove: person(influent0).
Let us try Prolog on this goal.
Prolog succeeds!
End of Case 2.
End of Proof.

We actually handle a seemingly more general class of problems than presented above, by allowing existential variables in the formula conclusion, precisely, problems of the form:

$$\text{comp}(P) \cup \text{contradict}(P) \models \forall \overline{X} \left[ \alpha \Rightarrow \exists \overline{Y} \beta \right]$$
where \( a \) and \( b \) are atoms (starting with predicates of possibly different arities), \( \overline{X} \) is a vector of all distinct variables occurring in \( a \), and \( \overline{Y} \) is a vector of all distinct variables occurring in \( b \) but not in \( a \). It is easy to see that this form is amenable to the simpler form previously presented

\[
\text{comp}(P) \cup \text{contradict}(P) \vdash \forall \overline{X} [p(\overline{X}) \Rightarrow q(\overline{X})]
\]

by adding to \( P \) two new auxiliary predicates \( p \) and \( q \), defined in terms of \( \alpha \) and \( \beta \).

The essence of our algorithm is the inference system below, which uses the following notations:

\( C \), the context, is a multiset of ground atoms, or equalities among constants or variables; \( \alpha, \beta \) denote atoms or equalities whose arguments are either variables or constants; \( \overline{X}, \overline{Y} \), denote vectors of distinct variables; \( \overline{A}, \overline{B} \) denote vectors of distinct constants; \( U, V, W \) denote either constants or variables, and \( \overline{U} \) denotes a vector of (not necessarily distinct) variables or constants (possibly mixed).

\( t[u/v] \) denotes the result of replacing \( u \) with \( v \) in \( t \), where \( t \) is a formula or a context, and either each one of \( u \) and \( v \) stands for a constant or variable, or \( u \) and \( v \) stand for vectors of same length, with \( u \) a vector of distinct items.

\( u=v \) denotes the multiset of (coordinate to coordinate) equalities, if \( u \) and \( v \) stand for vectors of equal length.

\( \nu(u_1, ..., u_n) \), resp. \( \kappa(u_1, ..., u_n) \), is the set of variables, resp. constants, occurring in one of \( u_1, ..., u_n \), where each \( u_i \) is a context, an atom or a vector of variables or constants.

Furthermore, following the sequent notation of Gallier [GALL86], in a sequent of the form \( C \vdash D \), \( C \) is interpreted as a conjunction, and \( D \) as a disjunction. However, in our application, \( D \) is always reduced to a single formula.

\[
\begin{align*}
(i) & \quad C \vdash (\alpha \Rightarrow \beta)[\overline{X}/\overline{A}] \quad \text{if } \nu(\overline{X}) = \nu(\alpha), \nu(\overline{Y}) = \nu(\beta) - \nu(\alpha), \kappa(\overline{A}) \cap \kappa(\alpha, \beta, C) = \emptyset ; \\
(ii) & \quad C, \alpha \vdash \beta \quad \text{if } \alpha \text{ is closed} ; \\
(iii) & \quad C, X = \overline{U} \vdash \ell_1[\overline{X}/\overline{U}] \quad ... \quad C, X = \overline{U} \vdash \ell_n[\overline{X}/\overline{U}] \quad \text{if } \nu(\overline{X}) \cap \nu(\overline{U}, C) = \emptyset ; \\
(iv) & \quad C, \beta_1[\overline{Y}/\overline{B}, \overline{X}/\overline{A}] \vdash \alpha \quad ... \quad C, \beta_n[\overline{Y}/\overline{B}, \overline{X}/\overline{A}] \vdash \alpha \quad \text{if } \kappa(\overline{A}) \cap \kappa(\overline{B}) = \emptyset ; \\
(v) & \quad C[A/B] \vdash \alpha[A/B] ; \quad (vi) \quad C, U = V, V = U \vdash \alpha ; \\
(vi) & \quad C, U = V, V = W, U = W \vdash \alpha ; \\
(vii) & \quad C, U = V, V = W \vdash \alpha ; \\
(viii) & \quad C, p(A), q(A) \vdash \alpha \quad \text{if } p \neq q \text{ and neither } p, \text{ nor } q, \text{ is defined by a clause in } C.
\end{align*}
\]
Rule (i) is a specialization of a method for proving a universally quantified formula: in order to prove $\forall \overline{X} \alpha$, prove the instance $\alpha[\overline{X}/\overline{A}]$, where $\overline{A}$ consists of new constants not submitted to any assumption. Rule (ii) is a restricted version of "implication introduction", limited to implications of atoms (with a ground premise). Prolog inference is represented by rule (iii). Rule (iv) corresponds to predicate unfolding. Rule (v) corresponds to equality substitution restricted to constants; rules (vi) and (vii) are respectively symmetry and transitivity. Rule (viii) implements the contradict(P) axioms.

The initial sequent to be proved is presented under the form:

$$ [P \cup \text{comp}(P)]^* \vdash \forall \overline{X} \{\alpha \Rightarrow \exists \overline{Y} \beta\} $$

where the $*$ notation on the left side should be understood as yielding an infinite number of clause variants in $P$ or definitions in $\text{comp}(P)$. Thus, removing the clause or definition used from the context in the subgoals of rules (iii) and (iv) does not lose completeness!

It is straightforward to check that our inference system is correct, that is, if

$$ [P \cup \text{comp}(P)]^* \vdash \forall \overline{X} \{\alpha \Rightarrow \exists \overline{Y} \beta\} $$

with $v(\overline{X})=v(\alpha)$ and $v(\overline{Y})=v(\beta)-v(\alpha)$, then

$$ \text{comp}(P) \cup \text{contradict}(P) \models \forall \overline{X} \{\alpha \Rightarrow \exists \overline{Y} \beta\} $$

We have implemented this inference system in Prolog III [COLM91], although we have not used constraints. Termination of the algorithm is guaranteed by the use of a Prolog meta-interpreter endowed with a goal stack preventing cyclic executions due to recursive predicates and by blocking recursive unfolding: we maintain a list of assumed facts, together with a history of predicates for each fact, which remembers all predicates used for unfolding some of its ancestors.

Applying this inference system to the politician example, we deduce the partial order already pictured by the following graph:

```
\begin{center}
\begin{tikzpicture}
  \node (person) at (0,0) {person};
  \node (influent) at (-1,-1) {influent};
  \node (civilian) at (1,-1) {civilian};
  \node (politician) at (0,-2) {politician};
  \draw[->] (person) -- (influent);
  \draw[->] (person) -- (civilian);
  \draw[->] (influent) -- (politician);
  \draw[->] (civilian) -- (politician);
\end{tikzpicture}
\end{center}
```

Figure 3. The partial order for the politician example.

A family of examples that are swiftly handled consists in rediscovering an inheritance hierarchy from its clausal translation. For instance, given the animal hierarchy adapted from [CONR87].

```
\begin{verbatim}
animal(X) :- herbivorous(X).
animal(X) :- carnivorous(X).
herbivorous(X) :- cow(X).
herbivorous(X) :- omnivorous(X).
carnivorous(X) :- cat(X).
carnivorous(X) :- omnivorous(X).
omnivorous(X) :- human(X).
\end{verbatim}
```

1262
and the additional clause

\[
\text{herb\_and\_carn}(X) :\neg \text{ herbivorous}(X), \text{ carnivorous}(X).
\]

Our system proves the equivalence of the omnivorous and herb\_and\_carn concepts, and rule (viii) is effectively applied to discard the case of an animal being a cow and a cat altogether as shown below in the proof of the goal omnivorous\_herb\_and\_carn.

\[
\text{implies(herb\_and\_carn}(X),\text{ omnivorous}(X));
\]

We are asked to prove this: for all \( X \), \( \text{herb\_and\_carn}(X) \Rightarrow \text{omnivorous}(X) \).

Let \( \text{herb\_and\_carn0} \) be a new constant.
Assume: \( \text{herb\_and\_carn(herb\_and\_carn0)} \).
We must prove: \( \text{omnivorous(herb\_and\_carn0)} \).
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: \( \text{herb\_and\_carn(herb\_and\_carn0)} \).
There is one case to consider:

Case 1.
Assume: \( \text{herbivorous(herb\_and\_carn0)} \),
\( \text{carnivorous(herb\_and\_carn0)} \).
We must prove: \( \text{omnivorous(herb\_and\_carn0)} \).
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: \( \text{herbivorous(herb\_and\_carn0)} \).
There are two cases to consider:

Case 1.
Assume: \( \text{cow(herb\_and\_carn0)} \).
We must prove: \( \text{omnivorous(herb\_and\_carn0)} \).
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: \( \text{carnivorous(herb\_and\_carn0)} \).
There are two cases to consider:

Case 1.
Assume: \( \text{cat(herb\_and\_carn0)} \).
\( \text{cat(herb\_and\_carn0)} \) contradicts previous assumption \( \text{cow(herb\_and\_carn0)} \)!
End of Case 1.
Case 2.
Assume: \( \text{omnivorous(herb\_and\_carn0)} \).
We must prove: \( \text{omnivorous(herb\_and\_carn0)} \).
Let us try Prolog on this goal.
Prolog succeeds!
End of Case 2.
End of Case 1.

Case 2.
Assume: \( \text{omnivorous(herb\_and\_carn0)} \).
We must prove: \( \text{omnivorous(herb\_and\_carn0)} \).
Let us try Prolog on this goal.
Prolog succeeds!
End of Case 2.
End of Case 1.
End of Proof.

Another interesting example is the definition of the transitive closure \( r^+ \), and transitive/reflexive closure \( r^* \) of a binary relation \( r \):
Our system proves $r \leq r^+ \leq r^*$, and finitely fails on wrong goals such as $r^* \leq r^+$. On this last wrong goal, our system displays the following output:

\[
\text{implies}(r^*(X,Y), r^+(X,Y))::
\]

We are asked to prove this: for all $X,Y$, $r^*(X,Y) \Rightarrow r^+(X,Y)$.

Let $r^0$ be a new constant, $r^1$ a new constant.
Assume: $r^*(r^0,r^1)$.
We must prove: $r^+(r^0,r^1)$.
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: $r^*(r^0,r^1)$.
There are two cases to consider:
Case 1.
Assume $r^0 = r^1$.
We must prove: $r^+(r^0,r^0)$.
Let us try Prolog on this goal.
Prolog fails!
We have nothing to unfold.
Failure of Case 1.
Discarding Case 2.

Proof Failed.

If we replace $r^+$ with $r^+$, and $r^*$ with $r^*$ in the above clauses (leaving $r$ itself unchanged), we obtain a new equivalent definition of $r$ transitive and transitive/reflexive closures, that is:

\[
r^+(X,Y) \quad : \quad r(X,Z), r^*(Z,Y).
\]

\[
r^*(X,X) 
\]

\[
r^*(X,Y) \quad : \quad r^+(X,Y).
\]

Proving that $r^* \leq r^*$ [resp. $r^* \leq r^*$] and $r^+ \leq r^+$ [resp. $r^+ \leq r^+$] requires adding the following induction rule to our inference system (Figure 2):

\[
C, p(\bar{A}), \beta_1[\bar{V}/\bar{B}, \bar{X}/\bar{A}] \vdash q(\bar{A})
\]

\[
\vdots
\]

\[
C, p(\bar{A}), \beta_n[\bar{V}/\bar{B}, \bar{X}/\bar{A}] \vdash q(\bar{A})
\]

\[
C, p(\bar{A}), (p(\bar{V}_1), q(\bar{V}_1), \ldots, p(\bar{V}_i), q(\bar{V}_i), \gamma_1, \ldots, \gamma_k)[\bar{V}/\bar{B}, \bar{X}/\bar{A}] \vdash q(\bar{A})
\]

\[
\vdots
\]

\[
C, p(\bar{A}), (p(\bar{Z}_1), q(\bar{Z}_1), \ldots, p(\bar{Z}_j), q(\bar{Z}_j), \delta_1, \ldots, \delta_m)[\bar{Y}/\bar{B}, \bar{X}/\bar{A}] \vdash q(\bar{A})
\]

Note that variables from $\bar{V}$, $\ldots$, $\bar{Z}$ vectors are necessarily variables from either $\bar{X}$ or $\bar{Y}$, and $v(\bar{X}) \cap v(\bar{Y}) = \emptyset$ by definition of $\text{comp}(P)$. 

1264
The proof below is triggered by our "impliesI" predicate, which implements the induction principle, needed for proving the goal $r^* \leq r^*$.

\textit{impliesI}(r^*(X,Y), r^*(X,Y)):
We are asked to prove this: $r^*$ inherits from $r^*$ according to least fixed point inclusion.

Let $r^*0$ be a new constant,
$r^*1$ be a new constant.
Assume: $r^*(r^*0, r^*1)$.
We must prove: $r^*(r^*0, r^*1)$.
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: $r^*(r^*0, r^*1)$.

There are two cases to consider:

Case 1.
Assume $r^*0 = r^*1$.
We must prove: $r^*(r^*0, r^*0)$.
Let us try Prolog on this goal.
Prolog succeeds!

End of Case 1.

Case 2.
Assume: $r^+(r^*0, r^*1)$.
We must prove: $r^*(r^*0, r^*1)$.
Let us try Prolog on this goal.
Prolog fails!
Let us unfold our assumption: $r^+(r^*0, r^*1)$.

There is one case to consider:

Case 1.
Let $r2$ be a new constant.
Assume: $r(r^*0,r2)$,
$r^*(r2,r^*1)$.
Assume by induction: $r^*(r2,r^*1)$.
We must prove: $r^*(r^*0, r^*1)$.
Let us try Prolog on this goal.
Prolog succeeds!

End of Case 1.

End of Case 2.

End of Proof.

There are two problems with this induction rule, that we must acknowledge. The first one is that we no longer prove a logical implication when we use this rule. In other words, predicate inheritance has to be redefined, as we shall see in the next section. The second problem is that correctness is not granted: correctness requires some restriction on Datalog programs P, such as "some measure decreases in the recursive calls". For instance, with the following program P,

\begin{align*}
p(X) & : - p(X). \\
p(X) & : - a(X). \\
q(X) & : - b(X).
\end{align*}

there exists a proof (using rule (ix)) of "p inherits from q modulo P".

4. Inheritance as Least Fixed Point Inclusion

Recall our definition of inheritance w.r.t. P:

\[
\text{def} \quad p \triangleleft p q \equiv \text{comp}(P) \cup \text{contradict}(P) \models \forall \bar{X} [p(\bar{X}) \Rightarrow q(\bar{X})]
\]
This definition allows us to claim that our inference system (without rule (ix)) proves "p≤_P q" when it is able to derive the sequent

\[ [P∪\text{comp}(P)]* \models \forall X[p(X)⇒q(X)] \]

Consider again the transitive closure program P of the previous section, with r*, r+ and r*'*, r+'*. Our inference system (without rule (ix)) cannot prove \( r^*≤_P r'^* \), that is,

\[ \text{comp}(P)∪\text{contradict}(P) \models \forall X\forall Y[r^*(X,Y)⇒r'^*(X,Y)] \]

which is actually untrue! Note that contradict(P) is empty in this case. A possible model of the left hand side \text{comp}(P), which is not a model of the right hand side, interprets r as mathematical equality "=" on \{0,1\}, r* as always true on this domain, and r*' as the number partial order "≤" on this domain.

We need a less restrictive definition of inheritance, based on the least fixed point operator. A database on P is a finite subset I of the "universal Herbrand base \( H_P \)" of P ("universal" means that constants are taken from an infinite set), whose each fact uses an undefined predicate, and whose no two facts contradict each other, i.e., if \( a(u)∈I \) and \( b(u)∈I \) for some undefined sorts a and b, then \( a=b \). Our new definition is:

\[ p≤'_P q \overset{\text{def}}{=} \forall I \text{ database on } P \quad \text{Args}(\text{Den}(p,P∪I)) \subseteq \text{Args}(\text{Den}(q,P∪I)) \]

with

\[ \text{Den}(p,P) = \text{lf}(\text{T}_P∩(p(\tilde{A}) | p(\tilde{A})∈H_P)} \]
\[ \forall I∈H_P \quad \text{Args}(I) = \{\tilde{A} | \exists p(\tilde{A})∈I\} \]

It is straightforward to prove that

\[ p≤_P q ⇒ p≤'_P q \]

but the converse is untrue.

Our intention, with our full inference system (including some correct variant of rule (ix) yet to be specified), is to establish inheritance according to this second definition.

5. Decidability of Inheritance(s)

The problem of deciding whether \( p≤'_P q \) holds, given a Datalog program P and two predicates p and q, of same arity, used in P, is closely related to the "Datalog query containment" problem [SHM93], although establishing the exact relationship would require a more detailed analysis than the intuitive overview we propose here.

The main differences between Shmueli problem and ours are:

(i) Shmueli problem refers to two Datalog programs, one for each query, whereas we deal with one program only;

(ii) The semantics of a Datalog query are given an operational flavor (SLD-derivation) whereas our formulation is in terms of a least fixed point;

(iii) There is no "undefined sort disjointedness" assumption in Shmueli problem, whereas we assume the databases to satisfy this restriction.

The difference (i) is unimportant, as it would be easy to merge both programs into one, by appropriate renaming of certain predicate symbols; (ii) may not be critical, because of the relationship between various Prolog semantics, see
(iii) may be alleviated by defining each undefined sort a in a program with a clause

\[ a(X) :\leftarrow a'(X,X). \]

where \( a' \) is a new predicate symbol of arity 2, so that our restriction on databases becomes void.

Shmueli has proved that "Datalog query containment" is undecidable [SHMU93]: the proof is by reduction to the context free language inclusion problem [HOPC79], and uses predicates of arity\geq 2. Li and Gloess have recently proved that even with arity restricted to 2 or less, the problem remains undecidable [LI94]: the proof is obtained by adapting Shmueli encoding of CFGs.

Using our remark on (iii), and provided that differences (i) and (ii) can effectively be erased, the \( \psi'_{pq} \) problem appears as a generalization of the Datalog query containment problem, which may take separation of undefined sorts into account. Therefore our problem \( \psi'_{pq} \) is undecidable, because otherwise the Datalog query containment subproblem would be decidable.

In any case, restricting to defined predicates of arity 1, that is, defined sorts, still gives a useful subclass, which might be decidable, although we have not proved it yet. It seems that a larger decidable subclass, related to Büchi alternating automata [BUCH90, BUCH88], might exist, that would allow "tail recursive" definitions of features.

The decidability of \( \psi_{pq} \) is an other issue: note that the "skolemized" form of the negation of this formula may contain function symbols other than constants, due to existential quantifiers in \( \text{comp}(P) \). Hence the Herbrand universe is not finite and the problem not purely propositional.

6. Conclusion

We have presented a method of classification of predicates applicable to Datalog programs. This method is an exact method, based on natural deduction. It thus differs from methods based on abstract interpretation, using an approximation of predicate denotations, which do not always yield correct results when applied to this kind of problem. It also seems simpler than the latter methods, as it directly works on the logic program (or its completed version) rather than on a type equation system derived from the program. It should be noted however that our method does not apply to programs with function symbols.

The method is implemented in Prolog III, using less than one hundred clauses (and no constraints)\(^1\). Decidability issues have been briefly discussed, and require additional work, as well as correctness of the induction rule, and completeness issues. The method may be applied to query or program optimization, in the context of databases, or program debugging. Elegance and robustness, rather than efficiency, have guided our Prolog implementation; however, the system runs reasonably fast for practical purposes.

Acknowledgement

We thank French D.R.E.T. for financial support through grant N°90-217, CONACyT (Mexico) for financial support to Miguel-Angel Oros. This work is partially supported by the French Project GASP. We thank Baudouin Le Charlier for an enlightening discussion on abstract interpretation, Bruno Courcelle and Mike O'Donnell for hints on decidability issues, and Jean-Marc Andréoli for helping

\(^1\) Our system, called "Concept", is available through the network by writing to the authors.
us clarify the definition of inheritance. We thank Christine Solnon for useful comments on our work, and for signalling bugs and suggesting examples.

References


