Fundamenta Informaticae 28 (1996) 23–36
IOS Press

Meta-variables in Logic Programming, or in Praise of Ambivalent Syntax

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands and
Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Rachel Ben-Eliyahu
Mathematics and Computer Science Department
Ben-Gurion University of the Negev
Beer-Sheva 84105
Israel

Abstract. We show here that meta-variables of Prolog admit a simple declarative interpretation. This allows us to extend the usual theory of SLD-resolution to the case of logic programs with meta-variables, and to establish soundness and strong completeness of the corresponding extension of the SLD-resolution. The key idea is the use of ambivalent syntax which allows us to use the same symbols as function and relation symbols.

We also study the problem of absence of run-time errors in presence of meta-variables. We prove that this problem is undecidable. However, we also provide some sufficient and polynomial-time-decidable conditions which imply absence of run-time errors.

Keywords: meta-variables, ambivalent syntax, Prolog programs, soundness and completeness, absence of errors.

1. Introduction

One of the unusual features of Prolog is the use of variables in the positions of atoms, both in the queries and in the clause bodies. Such a use of a variable is called a *meta-variable*. Meta-variables, when added to logic programs, allow us to extend their syntax in a simple way. For example, the program

```
or(X,Y) ← X.
or(X,Y) ← Y.
```

allows us to define disjunction, which can be declared as an infix relation ";", and subsequently used in another program or query, like in the following program ISO:

```
iso(void, void).
iso(tree(X,Left1,Right1), tree(X,Left2,Right2)) ←
   (iso(Left1,Left2),iso(Right1,Right2)) ;
   (iso(Left1,Right2),iso(Right1,Left2)).
```

which tests whether two binary trees are isomorphic.

Using meta-variables some other extensions of logic programming can be defined. For example, assuming for a moment that the cut "!" facility is present in the language, we can introduce an `if_then_else` predicate by means of the program

```
if_then_else(P, Q, R) ← P,!,Q.
if_then_else(P, Q, R) ← R.
```

and then define negation by the single clause

```
neg(X) ← if_then_else(X, fail, true).
```

where `true` is the query which immediately succeeds.

Other uses of meta-variables can be found in Prolog programs that solve puzzles. As an illustration consider the following puzzle from Smullyan [Smu94, page 23] and its solution in Prolog given in Casimir [Cas88]:

"Then there's my cook and the Cheshire Cat" continued the Duchess. "The Cook believes that at least one of the two is mad." What can you deduce about the Cook and the Cat?

It is assumed that every person is always saying the truth or always lying, and "mad" is to be identified here with "always lying".

```
is(truthful).
is(lying).

believes(Somebody, Sth) ←
   Somebody = truthful, Sth  ;
   Somebody = lying, ¬ Sth.

puzzle(Cook, Cat) ←
   is(Cook), is(Cat),
   believes(Cook, (Cook = lying ; Cat = lying)).
```

Here ";" denotes disjunction, as defined above, "¬" denotes negation and "=" is Prolog's built-in, called "is unifiable with" and defined by the single clause

```
X = X.
```

Executing the query `puzzle(Cook, Cat)` we get the desired answer:

```
?- puzzle(Cook, Cat).

Cat = lying,
Cook = truthful ;

no
```

Meta-variables are also useful when writing meta-interpreters, as they allow us to execute certain calls by "lifting" them to the system level — see for an instance the program considered in Example 6.1.

Prolog's approach to meta-programming, so the process of writing programs (like meta-interpreters) that use other programs as data, should be contrasted with that of the programming language Gödel of Hill and Lloyd [HL94], in which the data program is accessible indirectly, through its representation. In particular, there are no meta-variables in Gödel.

In this paper we provide theoretical foundations for the study of logic programs with meta-variables. We show that this seemingly illogical use of variables can be easily accounted for on a semantic level by means of ambivalent syntax which allows us to use the same symbols as function and relation symbols. More precisely, we first adopt a version of ambivalent syntax, then introduce a simple declarative semantics for logic programs with meta-variables, and establish soundness and strong completeness of the corresponding extension of the SLD-resolution.

Intuitively, a meta-variable is a "place holder" which before its selection should be replaced by an atom. Consequently, following Prolog, we stipulate that the selection of a meta-variable by the selection rule leads to a run-time error. We prove that – as expected – absence of run-time errors in presence of meta-variables is undecidable. However, we also provide some sufficient and decidable conditions which imply absence of run-time errors.

The use of the ambivalent syntax was first advocated in mathematical logic by Richards [Ric74], in the theory of logic programming by Kalsbeek [Kal93] and Jiang [Jia94], and in the programming languages area by Chen, Kifer and Warren[CKW89] in their logic programming language proposal HiLog.

In each of these references different versions of ambivalence are assumed. Our version just boils down to identification of function and relation symbols. This approach is related to that of De Schreye and Martens [DM92] in which overloading of function and relation symbols is used in order to provide semantics to meta-programs.

The results of our paper show that once ambivalent syntax is permitted, meta-variables admit a natural logical interpretation and can be easily reasoned about. Hence the title.

2. Syntax and Proof Theory

The step from meta-variables to ambivalent syntax is very natural. If we accept $solve(x) \leftarrow x$ as a syntactically legal clause, then it is natural to accept any instance of it as syntactically legal, as well. So for any non-variable term t in the assumed language $solve(t) \leftarrow t$ is a legal clause. Now the outermost symbol of t occurs in this clause both in the function symbol position and the relation symbol position. As t was arbitrarily chosen, we conclude that in presence of meta-variables the classes of function symbols and of relation symbols in the assumed language coincide, as soon as the closure under instantiation is assumed.

So assume from now on a fixed first-order language $\mathcal{L}$ such that the classes of function symbols and relation symbols in $\mathcal{L}$ coincide. In the sequel we consider queries and programs written in this subset. Their syntax extends the customary syntax of logic programs as both in queries and in the clause bodies we allow variables to appear in atoms positions. In such a context they will be referred to as meta-variables. From now on we write meta-variables in capital.

Formally, a *query*, is a possibly empty sequence of atoms or variables. In turn, a *clause* is a construct of the form $A \leftarrow \mathbf{B}$ where A is an atom and $\mathbf{B}$ is a query. Thus we do not allow variables to appear as a head of a clause. In this way we conform to Prolog syntax restrictions.

In the subsequent analysis we shall also use *resultants* which are constructs of the form $\mathbf{A} \leftarrow \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are queries. By an *expression* we mean an atom, query, resultant or a clause. Given a program P, we denote by $inst(P)$ the set of all instances of clauses of P and by $ground(P)$ the set of all ground instances of clauses of P. All the considered expressions and their instances are built out of symbols present in $\mathcal{L}$. If a query (respectively, a program) does not contain meta-variables, it is called a *logical query* (respectively, a *logical program*).

Further, $Var(E)$ denotes the set of variables occurring in the expression E. A *substitution* is a function from variables to terms with a finite domain; ϵ denotes the empty substitution. Given a substitution θ, the set of variables occurring in its domain or in the terms forming its range is denoted by $Var(\theta)$ and its restriction to the set of variables V by $\theta \mid V$. Finally, a substitution is called a *renaming* if it is a permutation of the variables from its domain. Recall that for every renaming θ there exists exactly one substitution θ^{-1} such that $\theta\theta^{-1} = \theta^{-1}\theta = \epsilon$.

The SLD-resolution in presence of meta-variables is defined as for logical programs (see e.g. Lloyd [Llo87]), with the exception that for every resolution step:

- the mgu employed acts now also on meta-variables,
- the selection of a meta-variable by the selection rule leads to an error.

The second condition is consistent with Prolog's interpretation of meta-variables.

It is useful perhaps to mention here that for more powerful versions of ambivalent logics, like the ones discussed in Kalsbeek and Jiang [KJ95], the unification algorithm has to be appropriately generalized. This is not so for the version of the ambivalent syntax we use here since it does not yield any syntactic changes on the atom level.

We now refer to SLD-resolution with the leftmost selection rule as *LD-resolution.*

Example 2.1. Consider the query $p(X), X$. When the program is $\{p(a) \leftarrow\}$, then the only (up to renaming) LD-derivation fails, when the program is $\{p(y) \leftarrow\}$ then the only LD-derivation ends in an error after one computation step, and when the program is $\{p(a) \leftarrow, a \leftarrow\}$ then the only LD-derivation is successful and yields the computed answer substitution $\{X/a\}$. This agrees with Prolog's interpretation.

Formally, we extend the SLD-resolution by stipulating that an SLD-derivation *ends in an error* when at the moment of evaluation the selected atom is a variable.

The following notion will be useful in our considerations.

Definition 2.1. Consider an SLD-derivation

$$Q_0 \xrightarrow{\theta_1} Q_1 \cdots \xrightarrow{\theta_n} Q_n \ldots \qquad (1)$$

Let for $i \geq 0$

$$R_i := Q_0\theta_1 \ldots \theta_i \leftarrow Q_i.$$

We call R_i the *resultant of level* i of (1).

In Section 4. we shall need the following lemma which involves resultants.

Lemma 2.1. [Disjointness] Consider an SLD-derivation of $P \cup \{Q\}$ with the sequence $d_1, \ldots, d_{n+1}, \ldots$ of input clauses used and with the sequence $R_0, \ldots, R_n, \ldots$ of resultants associated with it. Then for $i \geq 0$

$$Var(R_i) \cap Var(d_{i+1}) = \emptyset.$$

Proof:
It suffices to prove by induction on i that

$$Var(R_i) \subseteq Var(Q) \cup \bigcup_{j=1}^{i} (Var(\theta_j) \cup Var(d_j)), \qquad (2)$$

where $\theta_1, \ldots, \theta_n, \ldots$ are the substitutions used. The claim then follows by standardization apart (defined as in Lloyd [Llo87, page 41], so as the condition that each input clause d_i is variable disjoint with $(Var(Q) \cup \bigcup_{j=1}^{i-1}(Var(\theta_j) \cup Var(d_j))))$.
Base. $i = 0$. Obvious.

Induction step. Suppose (2) holds for some $i \geq 0$. Note that if $R_i = Q' \leftarrow \mathbf{A}, B, \mathbf{C}$ where B is the selected atom, and $d_{i+1} = H \leftarrow \mathbf{B}$, then $R_{i+1} = (Q' \leftarrow \mathbf{A}, \mathbf{B}, \mathbf{C})\theta_{i+1}$. Thus

$$\begin{aligned} & Var(R_{i+1}) \\ \subseteq\ & Var(R_i) \cup Var(\theta_{i+1}) \cup Var(d_{i+1}) \\ \subseteq\ & \quad \{\text{induction hypothesis (2)}\} \\ & Var(Q) \cup \bigcup_{j=1}^{i+1} (Var(\theta_j) \cup Var(d_j)). \end{aligned}$$

3. Semantics

As a next step in our study of logic programs with meta-variables we study their meaning. To this end we define the meaning of expressions, so a fortiori of queries and programs.

In general, it is not clear how to define the meaning of an expression in an interpretation of the language $\mathcal{L}$, because it is not clear how to define the meaning of meta-variables. We circumvent this problem by limiting our attention to a restricted classes of interpretations, the Herbrand interpretations. Then we discuss to what extent this restriction could be relaxed.

Formally, by a *Herbrand interpretation* we mean a set of ground atoms (or equivalently ground termms) in the language $\mathcal{L}$. By a *state* we mean a mapping assigning to each variable a ground term.

We now define a relation $I \models_\sigma E$ between a Herbrand interpretation I, a state σ and an expression E. Intuitively, $I \models_\sigma E$ means that E is true in I when its variables are interpreted according to σ.

- if X is a variable, then
$$I \models_\sigma X \text{ iff } \sigma(X) \in I,$$
- if A is an atom, then
$$I \models_\sigma A \text{ iff } A\sigma \in I,$$
- if $A_1, \ldots, A_n$ is a query, then
$$I \models_\sigma A_1, \ldots, A_n \text{ iff } I \models_\sigma A_i \text{ for } i \in [1, n],$$
- if $\mathbf{A} \leftarrow \mathbf{B}$ is a resultant, then
$$I \models_\sigma \mathbf{A} \leftarrow \mathbf{B} \text{ iff } (I \models_\sigma \mathbf{B} \text{ implies } I \models_\sigma \mathbf{A}).$$

In particular, if $H \leftarrow \mathbf{B}$ is a clause, then
$$I \models_\sigma H \leftarrow \mathbf{B} \text{ iff } (I \models_\sigma \mathbf{B} \text{ implies } I \models_\sigma H),$$
and for a unit clause $H \leftarrow$
$$I \models_\sigma H \leftarrow \text{ iff } I \models_\sigma H.$$

In this definition only the first statement is unusual. In the usual setting the condition on its right hand side does not make sense, and consequently can never succeed. But now the ambivalent syntax is assumed, so this statement is perfectly legal as every term is also an atom and consequently it can succeed.

Finally, given an expression E and a Herbrand interpretation I, we say that *E is true in I*, or *I is a Herbrand model of E*, and write $I \models E$, when for all states σ we have $I \models_\sigma E$. Note that the empty query is true in every Herbrand interpretation I. An interpretation I is called a *model* of a program P if all the clauses of P are true in I. When E is true in all Herbrand models of a program P, we write $P \models E$.

The following example hopefully clarifies the introduced notions.

Example 3.1. Suppose that $\mathcal{L}$ has only one constant (and 0-ary relation symbol) c, and one unary function (and relation) symbol *solve*. Let $P = \{solve(X) \leftarrow X\}$, and let $I = \{c, solve(c)\}$.

Then I is not a model of P, because $I \models solve(c)$ but not $I \models solve(solve(c))$. On the other hand for every $k \geq 0$, $J_k = \{solve^n(c) \mid n \geq k\}$ is a model of P, since every ground term of $\mathcal{L}$ is of the form $solve^n(c)$ for $n \geq 0$ and $solve^n(c) \in J_k$ implies $solve^{n+1}(c) \in J_k$. Also, the empty Herbrand interpretation is a model of P.

When trying to define the meaning of expressions in more general interpretations one has to clarify how to assign meaning to meta-variables. We see two possible approaches. The first one consists of considering term interpretations, that is interpretations whose universe consists of all terms. Then the appropriate notion of a state is that of a mapping assigning to each variable a (not necessarily ground) term and the first statement in the above definition of semantics still makes a perfect sense, as every term interpretation for the ambivalent language $\mathcal{L}$ can be identified with a set of terms. In our presentation we decided to limit our attention to Herbrand interpretations, as they are easier to understand and to deal with.

The second approach (suggested by a referee of an earlier version of this paper) consists of transforming each program and query into a logical program and a logical query in a first-order non-ambivalent language without meta-variables, and assign the meaning to the latter objects. To this end it suffices to replace every atom or meta-variable A by $holds(A)$, where *holds* is a new unary relation symbol.

From the proof theoretic point of view the transformed program and query behaves in an equivalent way to the original one with the important exception that errors due to the selection of a meta-variable X are mapped onto the selection of atoms of the form $holds(X)$. So this approach does not provide any means to prove absence of such errors. On the other hand, this type of transformations is useful when studying meta-interpreters.

From the semantic point of view this approach has a number of drawbacks. The reason is that it associates a meaning with a program *indirectly*, so the semantics of the programs like P in Example 3.1. is explained only in terms of a semantics of another, logical program. This approach makes "*holds*" a special relation symbol and does not blend well with the overwhelming body of results that follow the standard logic programming practice and define the meaning of a program *directly* in terms of the meaning of its relations.

For example, once a program transformation (for instance introduction of disjunction) introduces in a logical program a meta-variable, the semantics of the program changes even if the transformation ensures semantic equivalence. As a consequence, this approach does not support systematic program construction by means of programs transformation.

This in turn implies that this approach does not support modular program construction either. Indeed, in case of a program built out of modules it is customary to associate with a program semantics that is a function of the semantics of the underlying program modules. But this function has now to be changed once an underlying program module is a logical program and in the process of its refinement a meta-variable is introduced.

To cope with these problems one would have to use this "indirect" semantics for *all* programs, including the logical ones which is awkward and artificial.

It is useful to remark that semantics of programs that takes into account modularity is important both for program construction (see e.g. Brogi and Turini [BMPT94]) and for program verification (see e.g. Apt and Pedreschi [AP94]).

The program `ISO` of Section 1. suggests that one might get rid of meta-variables by unfolding. Indeed, by unfolding in `ISO` the call to the ";" relation we end up with a program without meta-variables. Unfortunately, this approach does not work in general. For example the meta-variables cannot be eliminated in this way from the other program from Section 1. or from the program P in Example 3.1.

We conclude this section by mentioning the following result which can be established by mimicking the corresponding proof for the case of (standard) SLD-resolution.

Theorem 3.1. [Soundness] *Suppose that there exists a successful SLD-derivation of* $P \cup \{Q\}$ *with the computed answer substitution* θ. *Then* $P \models Q\theta$.

4. Completeness

In this section we establish a completeness result. To this end we adjust the proof of strong completeness of SLD-resolution due to Stärk [Stä90]. We begin by introducing the following concept.

Definition 4.1. A finite tree whose nodes are atoms, is called an *implication tree w.r.t.* P if for each of its nodes A with the children $B_1, \ldots, B_n$, the clause $A \leftarrow B_1, \ldots, B_n$ is in $inst(P)$.

We say that an atom *has an implication tree w.r.t.* P if it is the root of an implication tree w.r.t. P. An implication tree is called *ground* iff all its nodes are ground.

In particular, for $n = 0$ we get that every leaf A of an implication tree is such that the unit clause $A \leftarrow$ is in $inst(P)$. The following lemma reveals the relevance of the implication trees for the semantics.

Lemma 4.1. The Herbrand interpretation

$$\mathcal{M}(P) := \{A \mid A \text{ has a ground implication tree w.r.t. } P\}$$

is a model of P.

Proof:
First note that for a Herbrand interpretation I, $I \models P$ iff $I \models ground(P)$. Now to show that $\mathcal{M}(P) \models ground(P)$ it suffices to prove that for all $A \leftarrow B_1, \ldots, B_n$ in $ground(P)$, $\{B_1, \ldots, B_n\} \subseteq \mathcal{M}(P)$ implies $A \in \mathcal{M}(P)$. But this translates into an obvious property of the ground implication trees. □

In fact, $\mathcal{M}(P)$ is the least Herbrand model of P, but this property is not needed here. This brings us to the following conclusion.

Corollary 4.1. Assume that the language $\mathcal{L}$ has infinitely many constants. Suppose that $P \models Q$. Then Q is a logical query and every atom in Q has an implication tree w.r.t. P.

Proof:
By Lemma 4.1. $\mathcal{M}(P) \models Q$. First note that Q is a logical query. Indeed, suppose otherwise. Then for some meta-variable X we have $\mathcal{M}(P) \models X$, so every constant c of $\mathcal{L}$ has a ground implication tree w.r.t. P. (Here the ambivalence of the syntax is used and the constants are "interpreted" as 0-ary relations.) So for every constant c of $\mathcal{L}$ there is a clause of P with c as its head. But P has only finitely many clauses, so this is impossible.

For the proof of the second property, let $x_1, \ldots, x_n$ be the variables of Q and $c_1, \ldots, c_n$ distinct constants of $\mathcal{L}$ which do not appear in P or Q. Let $\gamma := \{x_1/c_1, \ldots, x_n/c_n\}$. Then $Q\gamma$ is ground and $\mathcal{M}(P) \models Q\gamma$, so $Q\gamma \subseteq \mathcal{M}(P)$, that is every atom in $Q\gamma$ has a ground implication tree w.r.t. P. By replacing in these trees every occurrence of a constant c_i by x_i for $i \in [1, n]$ we conclude, by virtue of the choice of the constants $c_1, \ldots, c_n$, that every atom in Q has an implication tree w.r.t. P. □

Given a program P and a query Q, we now say that Q is *n-deep* if it is a logical query and every atom in Q has an implication tree w.r.t. P such that the total number of nodes in these implication trees is n. Then a query is 0-deep iff it is empty.

The following lemma relates two concepts of provability – that by means of implication trees and that by means of SLD-resolution.

Lemma 4.2. [Implication Tree] Suppose that $Q\theta$ is n-deep for some $n \geq 0$ and that all SLD-derivations of $P \cup \{Q\}$ via a selection rule $\mathcal{R}$ do not end in error.

Then there exists a successful *SLD*-derivation of $P \cup \{Q\}$ via $\mathcal{R}$ with the computed answer substitution η such that $Q\eta$ is more general than $Q\theta$.

Proof:
We construct by induction on $i \in [0, n]$ a prefix

$$Q_0 \overset{\theta_1}{\Longrightarrow} Q_1 \cdots \overset{\theta_i}{\Longrightarrow} Q_i$$

of an SLD-derivation of $P \cup \{Q\}$ via $\mathcal{R}$ and a sequence of substitutions $\gamma_0, \ldots, \gamma_i$, such that for the resultant $R_i := \mathbf{A}_i \leftarrow Q_i$ of level i

- $Q\theta = \mathbf{A}_i\gamma_i$,
- $Q_i\gamma_i$ is $(n-i)$-deep.

Then $Q_n\gamma_n$ is 0-deep, so Q_n is the empty query and consequently

$$Q_0 \overset{\theta_1}{\Longrightarrow} Q_1 \cdots \overset{\theta_n}{\Longrightarrow} Q_n$$

is the desired SLD-derivation, since $\mathbf{A}_n$ is then more general than $Q\theta$ and $\mathbf{A}_n = Q\theta_1 \ldots \theta_n$.

Base. $i = 0$. Define $Q_0 := Q$ and $\gamma_0 := \theta$.

Induction step. Let B be the atom or the meta-variable of Q_i selected by $\mathcal{R}$. By the assumption of the lemma B is an atom. Q_i is of the form $\mathbf{A}, B, \mathbf{C}$. By the induction hypothesis $B\gamma_i$ has an implication tree with $r \geq 1$ nodes. Hence there exists a clause $c := H \leftarrow \mathbf{B}$ in P and a substitution τ such that $B\gamma_i = H\tau$ and

$$\mathbf{B}\tau \text{ is } (r-1)\text{-deep.} \tag{3}$$

Let π be a renaming such that $c\pi$ is variable disjoint with Q and with the substitutions and the input clauses used in the prefix constructed so far. Further, let α be the union of $\gamma_i \mid Var(R_i)$ and $(\pi^{-1}\tau) \mid Var(c\pi)$. By the Disjointness Lemma 2.1. α is well-defined. α acts on R_i as γ_i and on $c\pi$ as $\pi^{-1}\tau$. This implies that

$$B\alpha = B\gamma_i = H\tau = H\pi(\pi^{-1}\tau) = (H\pi)\alpha,$$

so B and $H\pi$ unify. Define θ_{i+1} to be an mgu of B and $H\pi$. Then there is γ_{i+1} such that

$$\alpha = \theta_{i+1}\gamma_{i+1}. \tag{4}$$

Let $Q_{i+1} := (\mathbf{A}, \mathbf{B}\pi, \mathbf{C})\theta_{i+1}$ be the next resolvent in the SLD-derivation being constructed. Then $\mathbf{A}_i\theta_{i+1} \leftarrow Q_{i+1}$ is the resultant of level $i+1$. We have

$$\begin{array}{ll} & Q\theta \\ = & \quad \{\text{induction hypothesis}\} \\ & \mathbf{A}_i\gamma_i \\ = & \quad \{\text{definition of } \alpha\} \\ & \mathbf{A}_i\alpha \\ = & \quad \{(4)\} \\ & \mathbf{A}_i\theta_{i+1}\gamma_{i+1}, \end{array}$$

and

$$\begin{array}{ll} & Q_{i+1}\gamma_{i+1}, \\ = & (\mathbf{A}, \mathbf{B}\pi, \mathbf{C})\theta_{i+1}\gamma_{i+1} \\ = & \quad \{(4)\} \\ & (\mathbf{A}, \mathbf{B}\pi, \mathbf{C})\alpha \\ = & \quad \{\text{definition of } \alpha\} \\ & \mathbf{A}\gamma_i, \mathbf{B}\tau, \mathbf{C}\gamma_i. \end{array}$$

So $Q_{i+1}\gamma_{i+1}$ is obtained from $Q_i\gamma_i$ by replacing $B\gamma_i$, that is $H\tau$, by $\mathbf{B}\tau$. By the induction hypothesis and (3) we conclude that $Q_{i+1}\gamma_{i+1}$ is $(n-(i+1))$-deep. This completes the proof of the induction step. □

We can now prove the desired result.

Theorem 4.1. [Strong Completeness] *Assume that the language $\mathcal{L}$ has infinitely many constants. Suppose that $P \models Q\theta$ and that all SLD-derivations of $P \cup \{Q\}$ via a selection rule $\mathcal{R}$ do not end in error.*

Then there exists a successful SLD-derivation of $P \cup \{Q\}$ via $\mathcal{R}$ with the computed answer substitution η such that $Q\eta$ is more general than $Q\theta$.

Proof:
By the Corollary 4.1. $P \models Q\theta$ implies that $Q\theta$ is n-deep for some $n \geq 0$. The claim now follows by the Implication Tree Lemma 4.2. □

The assumption that the language $\mathcal{L}$ has infinitely many constants is necessary here. Indeed, suppose that $\mathcal{L}$ has only finitely many constants, say $c_1, \ldots, c_n$. Let P consist of the unit clauses $solve(c_1), \ldots, solve(c_n)$, and the clause $solve(solve(x)) \leftarrow solve(x)$, where $solve$ is a unary function and relation symbol (we make use here of the ambivalence of the syntax). Note that every ground term in $\mathcal{L}$ is of the form $solve^i(c_j)$ for some $i \geq 0$ and $j \in [1..n]$, and that every such term, viewed as an atom, belongs to every Herbrand model of P.

Take now the query $Q := solve(x)$. Note that $P \models Q\epsilon$. Also, all LD-derivations of $P \cup \{Q\}$ do not end in error. In fact, meta-variables are not used here. However, every successful LD-derivation of $P \cup \{Q\}$ yields a computed answer substitution η such that $Q\eta$ is of the form $solve(c_j)$ for some $j \in [1..n]$, so not more general than $Q\epsilon$.

This is in contrast to the classical theory of the SLD-resolution where the strong completeness does not depend on the underlying language. It is useful to understand the reasons for this difference.

In the classical case of logical programs and logical queries semantics is defined for arbitrary interpretations, whereas in presence of meta-variables only for Herbrand interpretations. Now, for logical programs and logical queries the truth in all interpretations is in general not equivalent to truth in all Herbrand interpretations but the equivalence does hold when the underlying language has infinitely many constants — see Maher [Mah88]. So when infinitely many constants are present in the language, the completeness theorem for logical programs and logical queries does hold when only Herbrand interpretations are used. Thus the above theorem extends this version of the completeness theorem to programs and queries in presence of meta-variables.

It is worthwhile to note that when the semantics based on all term interpretations is used, then the corresponding completeness result does not require that the underlying language has infinitely many constants. The proof of this result is analogous to the proof of the Strong Completeness Theorem 4.1. and is omitted. In fact, in the case of logical programs and logical queries the truth in all interpretations is always equivalent to truth in all term interpretations — see Falaschi et al. [FLMP89], and this results extends to programs and queries in presence of meta-variables.

Also, when the other approach to semantics of programs and queries discussed at the end of Section 3. is used, so the one involving the translation by means of the relation symbol *holds*, the corresponding completeness result does not depend on the assumptions about the underlying language. This is the consequence of the fact that the semantics of the translated program and translated query is given in terms of arbitrary interpretations and not only Herbrand interpretations.

The assumption that the language $\mathcal{L}$ of programs has infinitely many constants sounds perhaps artificial. However, at a closer look it is quite natural. For example, any Prolog manual defines *infinitely* many constants. Of course, in practice only finitely many of them can be written or printed. But even in the case of one fixed program arbitrary queries can be posed, and in these queries arbitrary constants can appear. So when studying behaviour of a program, it is natural to assume a language in which all these constants are present.

5. Absence of Errors

When studying SLD-resolution in presence of meta-variables it is natural to seek conditions that ensure that the SLD-derivations do not end in error. It is particularly of interest when studying correctness of Prolog programs that use meta-variables, like the `ISO` program discussed in Section 1. The following result shows that this property is in general undecidable.

Theorem 5.1. *For some logical program P the following property is undecidable:*

a query Q is such that all LD-derivations of $P \cup \{Q\}$ do not end in error.

Proof:
Below M_P denotes the least Herbrand model of a program P and B_P the Herbrand base determined by P. By the strong completeness of SLD-resolution we have for every program P and a ground atom A:

$$A \in M_P \text{ iff there exists a successful LD-derivation of } P \cup \{A\},$$

so

$$\begin{aligned} A \in B_P - M_P & \text{ iff no successful LD-derivation of } P \cup \{A\} \text{ exists} \\ & \text{ iff all LD-derivations of } P \cup \{A, X\} \text{ do not end in error,} \end{aligned}$$

where X is a meta-variable. Thus to prove the theorem it suffices to exhibit a program P for which the set M_P, and consequently the set $B_P - M_P$ is undecidable. Now, this is the contents of Corollary 4.7 in Apt [Apt90]. This completes the proof. □

6. Sufficient Conditions for Error-Free Computations

In this section we provide sufficient conditions on programs and queries that imply absence of errors of the kind defined in the previous sections. We also show that these sufficient conditions can be checked in time polynomial in the size of the program and the query.

We start by introducing meta-modes. Meta-modes indicate how the arguments of a relation should be used. Intuitively, in order to prevent run-time errors, we should avoid having a variable as the i'th argument of the query $p(...)$ if i is in the meta-mode for p.

Definition 6.1. [meta-mode] Consider an n-ary relation symbol p. A *meta-mode* for p, m_p, is a subset of $\{1, ..., n\}$. By a *meta-moding* for a program P we mean a collection of modes, one for each relation symbol in the language $\mathcal{L}$ and such that $m_p = \emptyset$ for all relation symbols p not in P.

Sometimes we shall say just mode (resp. moding) instead of meta-mode (resp. meta-moding).

Example 6.1. Consider the following program `SOLVE` from Sterling and Shapiro [SS86, pages 307-308], where `solve(Query)` succeeds whenever `Query` is deduced from the Prolog program defined by a binary relation symbol `clause`. To avoid some uninteresting syntax complications we assume here that each program clause $H \leftarrow \mathbf{B}$ is represented by the atom `clause(H, Bs)`, where `Bs` is the list of atoms forming $\mathbf{B}$. We also assume that the relation symbol `system` defines the system predicates.

```
solve([]).
solve([A | Bs])) ← solve(A), solve(Bs).
solve(A) ← system(A), A.
solve(A) ← clause(A, Bs), solve(Bs).
```

Below we consider the following meta-moding for this program: $m_{solve} = \{1\}$, $m_p = \emptyset$ for all other relation symbols of $\mathcal{L}$.

We now define when a variable is considered to be a meta-variable in a query. From now on assume a fixed moding for each considered program.

Definition 6.2. [The relations $\leadsto$ and $\leadsto^*$] Consider an atom $A := p(t_1, \ldots, t_n)$. Suppose that $i \in m_p$. Then we write $A \leadsto t_i$. Due to the ambivalent syntax $\leadsto$ can be viewed as a binary relation both on terms and on atoms. $\leadsto^*$ denotes the transitive, reflexive closure of $\leadsto$.

Definition 6.3. [meta-variable in a query]

- A variable X is a *meta-variable in an atom* A if $A \leadsto^* X$.
- A variable X is a *meta-variable in a query* if it occurs in it as a meta-variable or it is a meta-variable in some of its atoms.

Intuitively, $A \leadsto^* X$ holds if in the parse tree for A an occurrence of the variable X can be reached from the root via a path with only "meta-moded" links.

Example 6.2. For the moding given in Example 6.1., X is a meta variable in the queries `solve(solve(X))` and `system(X),X`, but X is not a meta-variable in the query `solve(p(X))`, where p is a relation symbol different from `solve`.

To deal with absence of errors in presence of meta-variables we now introduce the notion of well-meta-modedness.

Definition 6.4. [well-meta-moded (wmm)]

- A query Q is called *well-meta-moded* (in short *wmm*) if no variable is a meta-variable in Q.
- A clause $A \leftarrow Q$ is called *well-meta-moded* if for every meta-variable X in Q we have $A \leadsto X$.
- A program is called *well-meta-moded* if every clause of it is.

The theorem below explains our interest in the notion of well-meta-modedness. We need the following lemma.

Lemma 6.1. An SLD-resolvent of a well-meta-moded query and a well-meta-moded clause that is variable disjoint with it, is well-meta-moded.

Proof:
First note that an instance of a wmm query is wmm. Indeed, if $A\theta \leadsto^* X$ then either A is a meta-variable or $A \leadsto^* X$ or for some binding $Y/s \in \theta$ both $A \leadsto^* Y$ and $s \leadsto^* X$.

Suppose now that a wmm query Q is (successfully) resolved with the wmm clause $c := p(t_1, ..., t_k) \leftarrow B$. Let A be the selected atom in Q. For some terms $s_1, ..., s_k$ $A := p(s_1, ..., s_k)$. Let X be a meta-variable in B. Since c is wmm, for some $i \in [1, k]$ we have $X = t_i$ and $i \in m_p$. Since Q is wmm, s_i is a term having no meta-variables. Hence when c is instantiated with an mgu of A and $p(t_1, ..., t_k)$ all the meta-variables in B are replaced with terms having no meta-variables.

This implies that the SLD-resolvent is wmm. □

Theorem 6.1. [Absence of Errors] *If P and Q are well-meta-moded then all SLD-derivations of $P \cup \{Q\}$ are error-free.*

Proof:
It is an immediate consequence of Lemma 6.1. □

We now turn to complexity issues. First note the following result.

Theorem 6.2. *Let μ be a moding for a program P. There exists an algorithm which checks whether P (resp. a query Q) is wmm w.r.t. μ in time polynomial in the size of P (resp. Q).*

Proof:
The size any moding for a program P is polynomial in the size of P. In fact, it is $O(nk)$, where n is the number of relation symbols and k the maximum arity. Hence, the relations $\leadsto$ and $\leadsto^*$, defined in Definition 6.2. can be computed in time which is polynomial in the size of P, and the number of pairs in these relations is polynomial in the size of P.

Deciding whether a variable is a meta-variable in some query (Definition 6.3.) can be done in time linear in the size of the relation $\leadsto^*$. So for each clause and for each query we can decide whether it is well-meta-moded (Definition 6.4.) in time polynomial in the size of the relations $\leadsto$ and $\leadsto^*$. Hence we can decide whether a program P (resp. a query Q) is wmm with respect to some moding in time which is polynomial in the size of P (resp. Q). □

This shows that the conditions of the Absence of Errors Theorem 6.1. can be checked in polynomial time.

Frequently, a moding that assigns to each n-ary relation symbol of the program the mode $\{1, ..., n\}$ will make the program well-meta-moded, but then the class of well-meta-moded queries becomes too restrictive. Hence the motivation for a *minimal* meta-moding.

Definition 6.5. A moding μ for a program P is a *good meta-moding* for P iff P is well-meta-moded with respect to μ and μ is minimal. That is, there is no other moding μ' such that P is well-meta-moded w.r.t. μ' and for some relation symbol p, $m'_p \in \mu'$, $m_p \in \mu$ and $m'_p \subset m_p$.

Example 6.3.

(i) The moding provided in Example 6.1. is a good meta-moding for the program SOLVE. By the Absence of Errors Theorem 6.1. applied to the program SOLVE and the query solve(p(X)) we conclude that the SLD-derivations of SOLVE ∪ {solve(p(X))} are error-free. This conclusion cannot be drawn for the query solve(solve(X)) which is not wmm. In fact, an SLD-derivation of SOLVE ∪ {solve(solve(X))} that repeatedly uses the third clause of SOLVE ends in an error.

(ii) The program which consists of the single clause

```
p(X)  ←  q(X), Y
```

does not have a good meta-moding.

(iii) Consider the following program P:

```
p(X,Y,Z)  ←  q(X,Y), Z.
q(X,Y)  ←  r(Y), X
```

Let μ be a moding such that $m_p = \{1,3\}$, $m_q = \{1\}$, and $m_r = \emptyset$. Then μ is a good meta-moding for P.

The query p(a, b, Z) is not wmm w.r.t. μ, whereas the query p(a, Y, r(X)) is wmm w.r.t. to μ. The query X is not wmm w.r.t. any moding. By the Absence of Errors Theorem 6.1. all SLD-derivations of P ∪ {p(a, Y, r(X))} are error-free.

We conclude with the following result concerning good meta-modings.

Theorem 6.3. *There exists an algorithm which checks whether a program P has a good meta-moding and provides such moding if it exists. This algorithm runs in time polynomial in the size of P.*

```
Good-meta-moding
    Input: A program $P$
    Output: If $P$ has a good meta-moding, such a moding will be the output.
    Otherwise $false$ is returned.
Let $p_1, ..., p_n$ be all the relation symbols in $P$.
for $i := 1$ to $n$ do $m_{p_i} := \emptyset$;
$\rightsquigarrow^* := \{(X, X) \mid X$ is a variable in P$\}$;
change :=$true$; fail := $false$;
while change and not fail do

    change := $false$;
    for each clause $p(t_1, ..., t_k) \leftarrow Q$ in $P$ do
        for each $X$ which is currently a meta-variable in $Q$ do
         if for some $1 \leq j \leq k$ $t_j = X$ then
           if $j \notin m_p$ then
           begin
            $m_p := m_p \bigcup \{j\}$;
            change := $true$;
           end;
         else fail:=$true$;
        endfor;
    endfor;
    compute $\rightsquigarrow^*$ according to the current values of $m_{p_1}, ..., m_{p_n}$;
endwhile;
for each relation symbol $p$ not in $P$ do $m_p = \emptyset$;
if not fail then return $m_{p_1}, ..., m_{p_n}$ else return $false$
```

Figure 1 Algorithm Good-meta-moding

Proof:
Consider the algorithm *Good-meta-moding* (in short, gmm) given in Figure 1 Suppose the input to algorithm gmm is some program P. First, note that the **while**-loop repeats at most $n * k$ times, where n is the number of relation symbols and k the maximum arity of any relation in P. Recall that the relation $\rightsquigarrow^*$ can be computed in time which is polynomial in the size of the program, and once this relation is given, testing whether a variable is a meta-variable in some query is also easy. Hence the algorithm runs in time which is polynomial in the size of the program. To verify that the algorithm indeed generates a correct output, note that the following invariants hold after each time the body of the **while**-loop is executed:

1. For every relation symbol p, if $j \in m_p$ then also j is in m_p in every other moding that makes P well-meta-moded,
2. If $fail = true$ then P has no good meta-moding.

The proof of the invariants is done by induction on i, the number of times the body of the **while**-loop was executed so far. Hence we have shown an algorithm which checks in polynomial time whether a program P has a good meta-moding and provides such moding if it exists. □

Acknowledgements

We would like to thank the referees of an earlier version of this paper and Marianne Kalsbeek for useful comments.

References

[AP94] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.

[Apt90] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Toplas*, 16(4):1361–1398, 1994.

[Cas88] R. Casimir. Is Prolog echt zo bijzonder. *Informatie*, 30(7/8):484–491, 1988. In Dutch.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North-American Conference on Logic Programming*, Cleveland, Ohio, October 1989.

[DM92] D. De Schreye and B. Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation. In A. Pettorossi, editor, *Proceedings Meta '92*, Lecture Notes in Computer Science 649, pages 192–204. Springer-Verlag, 1992.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis of metalogic programming: I. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 387–401. MIT Press, June 1994.

[Kal93] M. Kalsbeek. The vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, 1993.

[KJ95] M. Kalsbeek and Y. Jiang. A vademecum of ambivalent logic. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 27–56. The MIT Press, Cambridge, Massachusetts, 1995.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 348–357. The MIT Press, 1988.

[Ric74] B. Richards. A point of reference. *Synthese*, 28:431–445, 1974.

[Smu94] R. Smullyan. *Alice in Puzzle-land.* Penguin, Harmondsworth, 1994.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Stä90] R. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Computer Science Logic 89*, Lecture Notes in Computer Science 440, pages 382–383. Springer-Verlag, 1990.