PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

DIRECTIONS FOR META-PROGRAMMING

J.W. Lloyd

Department of Computer Science
University of Bristol
University Walk
Bristol BS8 1TR

Abstract

This paper contains a discussion of current research issues for meta-programming in Logic Programming. First, various difficulties with the way meta-programming is handled by Prolog are presented. It is shown that the cause of these problems can be traced to inadequacies in the way Prolog represents object programs at the meta-level and it is indicated how these problems can be overcome. A list of research problems in meta-programming is presented. The paper concludes with some remarks about the need to create improved Logic Programming languages.

1 Introduction

The quality of a programming language depends mainly on two factors: its expressiveness and its semantics. A programming language which is highly expressive allows programmers to easily and quickly write their programs at a conveniently high level. A programming language which has a simple and elegant mathematical semantics allows programmers to more easily verify and debug their programs and to be assured of the correctness of program transformations, optimisations, and so on.

Prolog's success is undoubtably due to its very high expressiveness. In a wide variety of application areas, programmers are able to get the job done more easily and quickly in Prolog than in other languages. Prolog's importance and widespread use is well justified by these advantages. However, Prolog's semantics (and by Prolog, we mean the practical programming language as it is embodied in currently available Prolog systems, not the idealised pure subsets studied in [Llo87], for example) is much less satisfactory. The problems with the semantics are numerous and well known: lack of occur check, unsafe negation, undisciplined use of cut, assert and retract, and so on. These so-called "impure" aspects of Prolog cause many practical Prolog programs to have no declarative semantics at all and to have unnecessarily complicated procedural semantics. This means that the verification, (systematic) construction, transformation, optimisation, and debugging of many Prolog programs is practically impossible.

The solution to these problems is to take more seriously the central thesis of Logic Programming, which is that

- a program is a (first order) theory, and
- computation is deduction from the theory.

It is crucially important that programs be interpreted directly as theories. When this is the case, they have simple declarative semantics and can be much more easily verified, transformed, debugged, and so on. Each of the impure aspects mentioned above causes difficulties precisely because it creates an impediment to the understanding of a program as a theory. A Prolog program which cannot be understood in some simple way as a theory has only a procedural semantics. This leaves us in no better position to understand the program than if it was written in a conventional procedural language.

Let us now concentrate on meta-programming and see what properties are desirable in a Logic Programming language for meta-programming and to what extent Prolog satisfies these properties. Although there is no clear consensus on precisely what meta-programming is about, its essential characteristic seems to be that a meta-program is a program which uses another program (the object program) as data. In any case, by any definition, meta-programming techniques underlie many of the applications of Logic Programming. For example, knowledge base systems consist of a number of knowledge bases (the object programs), which are manipulated by interpreters and assimilators (the meta-programs). Other important kinds of software, such as debuggers, compilers, and program transformers, are meta-programs.

There is now a vast literature on various aspects of meta-programming in Logic Programming. For example, papers concerned with the application of meta-programming to knowledge base systems include [Bow85], [BW85], [CFL*88], [LS87a], [MKK*83],

[OH88], [SB86], and [TF86]. The textbook of Sterling and Shapiro [SS86b] contains a discussion of the programming techniques relevant to meta-programming. Only a few papers discuss the theoretical foundations of meta-programming. These include [BK82], [Esh86], [HL88], [Sub88]. Other aspects of meta-programming are discussed in [Neu86], [SS86a], [SRP*88], and [Ven84]. A collection of recent papers on meta-programming is contained in [Llo88].

However, in spite of the fact that meta-programming techniques are widely and successfully used, the foundations of meta-programming and the meta-programming facilities provided by most currently available Prolog systems are by no means satisfactory. For example, on the theoretical side, some important representation (that is, naming) and semantic issues are normally glossed over. Furthermore, most currently available Prolog systems do not make a clear distinction between the object level and meta-level, do not provide explicit language facilities for representation of object level expressions at the meta-level, and do not provide important meta-programming software, such as partial evaluators.

As we show below, Prolog's meta-programming problems can be traced to the fact that it doesn't handle the representation requirements properly. A consequence of this is that it is not possible to (directly) interpret most Prolog meta-programs as theories and hence they do not have a declarative semantics. The most obvious symptom of this is with *var*, which has no declarative semantics at all in Prolog. However, the major point to be made in this paper is that once an appropriate representation is used, there are no theoretical impediments to obtaining a satisfactory declarative and procedural semantics for the meta-programming facilities of Prolog. Within the framework of the appropriate representation, a meta-program is a (typed) first order theory and the meta-logical predicates, such as *var*, have straightforward definitions, which provide simple and elegant declarative and procedural semantics.

We now point out precisely where Prolog's meta-programming problems lie. The first problem is concerned with an important and largely neglected representation issue, which is illustrated by one of the best known meta-programs, the standard *solve* interpreter. This interpreter consists of the following definition for *solve*

$$\begin{aligned}&solve(empty) \leftarrow\\ &solve(x\&y) \leftarrow solve(x) \wedge solve(y)\\ &solve(x) \leftarrow clause(x,y) \wedge solve(y)\end{aligned}$$

together with a definition for *clause*, which is used to represent the object program. For example, if the object program contains the clause

$$p(x,y) \leftarrow q(x,z) \wedge r(z,y)$$

then there is a corresponding clause of the form

$$clause(p(x,y),\ q(x,z)\&r(z,y)) \leftarrow$$

appearing in the definition of *clause*.

This interpreter is sometimes called the *vanilla* interpreter [SB86]. Many important meta-programs are extensions of one kind or another of the vanilla interpreter. See, for example, [SS86a], [SB86], and [SS86b].

However, the declarative meaning of the vanilla interpreter is by no means clear. The problem is that the variables in the definition of *clause* and the variables in the definition of *solve* intuitively range over different domains. (Informally, the variables in *clause* range over elements of the domain of the intended interpretation (based on a pre-interpretation J, say) of the object program, while the variables in *solve* range over conjunctions of J-instances of atoms.) Thus the intended interpretation is simply not a model of the program. This is not just a minor mathematical oddity. In particular, the problem certainly cannot be solved by simply asserting that each kind of variable is just a "logical variable". What is at stake here is whether it is possible to give a simple and precise semantics to the vanilla interpreter and other meta-programs. Without such a semantics, it is impossible, for example, to verify them or prove the correctness of transformations performed on them.

If the different kinds of variables are intended to range over different domains, then there is a clear solution. We should introduce types (also called sorts) into the language underlying the meta-program. This was called the *typed* representation in [HL88]. Then, for example, using an appropriately typed version of the vanilla interpreter, it is possible to prove its soundness and completeness for both the declarative and procedural semantics.

However, the typed representation introduces another problem related to the fact that an object level variable is represented by a meta-level *variable*. This leads to severe semantic problems with the meta-logical predicate *var* [SS86b]. With this representation, there seems to be no way of giving a declarative semantics to *var*. To see the difficulty, consider the goals

$$\leftarrow var(x) \wedge solve(p(x))$$

and

$$\leftarrow solve(p(x)) \wedge var(x)$$

If the object program consists solely of the clause $p(a)\leftarrow$, then (using the "leftmost literal" computation rule) the first goal succeeds, while the second goal fails.

These considerations lead to another representation scheme in which object level expressions are represented by ground terms at the meta-level. In such a representation, an object level variable is represented by a meta-level constant, say. This kind of representation, which was called the *ground* representation in [HL88], is a standard tool in mathematical logic. Using the ground representation, it is possible to give appropriate definitions for the meta-logical predicates of Prolog. It is also possible to give an interpreter which captures the procedural semantics of normal programs and goals given by SLDNF-resolution.

In the next section of this paper, we indicate how the representation problems can be solved. We describe the typed and ground representations, and briefly discuss two interpreters, one based on the typed representation and one based on the ground representation. We also indicate how the meta-logical predicates of Prolog can be defined. This section is substantially based on [HL88], which was written in collaboration with Pat Hill. In the third section, we discuss a variety of research problems in meta-programming in the hope of stimulating further research. We conclude with some remarks about the need to create improved Logic Programming languages.

2 Representation Issues

A *representation* [1] is a mapping from one language to another which is used to represent (that is, name) terms and formulas of the first language by terms in the second language. In this section, we outline two basic representations (the typed and ground representations), which are schemes for representing the quantifier-free formulas of a (type-free) language $\mathcal{L}$ in a typed language $\mathcal{L}'$. The typed representation seems to have been first defined explicitly in [HL88]. The ground representation is closely related to the kind of representation (usually called "Gödel numbering") employed in mathematical logic. The first mention of the ground representation in the Logic Programming literature seems to be in [Kow79].

First, we discuss the typed representation. $\mathcal{L}'$ has two types o and μ. Given a constant a in $\mathcal{L}$, there is a corresponding constant a' of type o in $\mathcal{L}'$. Given a variable x in $\mathcal{L}$, there is a corresponding variable x' of type o in $\mathcal{L}'$. Given an n-ary function symbol f in $\mathcal{L}$, there is a corresponding n-ary function symbol f' of type $o \times \cdots \times o \rightarrow o$ in $\mathcal{L}'$. Given an n-ary predicate symbol p in $\mathcal{L}$, there is a corresponding n-ary function symbol p' of type $o \times \cdots \times o \rightarrow \mu$ in $\mathcal{L}'$. We assume that the mappings $a \rightarrow a'$, $x \rightarrow x'$, $f \rightarrow f'$, and $p \rightarrow p'$ are all injective. The language $\mathcal{L}'$ has a constant *empty* of type μ. In addition, $\mathcal{L}'$ contains the function symbols &, if, and *not* of type $\mu \times \mu \rightarrow \mu$, $\mu \times \mu \rightarrow \mu$, and $\mu \rightarrow \mu$, respectively. Finally, $\mathcal{L}'$ has two predicate symbols *solve* and *clause* of type μ.

[1] A representation is not to be confused with the (slightly related) concepts of representable relation and representable function from mathematical logic. Note that the term "naming relation" rather than "representation" has more commonly been used in the Logic Programming literature [BK82], [Esh86], [Kow79].

If a is a constant, x is a variable, f is a function symbol, and p is a predicate symbol in $\mathcal{L}$, then we represent a by a', x by x', f by f', and p by p' in $\mathcal{L}'$. We define the representation of terms inductively. If $f(t_1, \ldots, t_n)$ is a term in $\mathcal{L}$, we represent $f(t_1, \ldots, t_n)$ by the term $f'(t'_1, \ldots, t'_n)$ of type o in $\mathcal{L}'$, where $t'_1, \ldots, t'_n$ are the representations of $t_1, \ldots, t_n$, respectively. If $p(t_1, \ldots, t_n)$ is an atom in $\mathcal{L}$, we represent $p(t_1, \ldots, t_n)$ by the term $p'(t'_1, \ldots, t'_n)$ of type μ in $\mathcal{L}'$. We also define the representation of quantifier-free formulas inductively. If the quantifier-free formulas F and G in $\mathcal{L}$ are represented by F' and G', respectively, then $\neg F$, $F \wedge G$, and $F \leftarrow G$ are represented by the terms $not(F')$, $F' \& G'$, and $F'\ if\ G'$, respectively, of type μ in $\mathcal{L}'$.

The typed normal program **V**, based on the language $\mathcal{L}'$, consists of the following definition for *solve*.

$solve(empty) \leftarrow$

$$\begin{aligned}\forall_\mu x, y\ \ & solve(x \& y) \leftarrow \\ & \quad solve(x) \wedge \\ & \quad solve(y)\end{aligned}$$

$$\begin{aligned}\forall_\mu x\ \ & solve(not(x)) \leftarrow \\ & \quad \neg solve(x)\end{aligned}$$

$$\begin{aligned}\forall_\mu x, y\ \ & solve(x) \leftarrow \\ & \quad clause(x\ if\ y) \wedge \\ & \quad solve(y)\end{aligned}$$

Given a normal program P, based on the language $\mathcal{L}$, the program $\mathbf{V}_P$ consists of the above program **V**, together with a clause of the form

$$\forall_o x'_1, \ldots, x'_k\ \ clause(A'\ if\ Q') \leftarrow$$

for every clause $A \leftarrow Q$, with variables $x_1, \ldots, x_k$, in P, and a clause of the form

$$\forall_o x'_1, \ldots, x'_k\ \ clause(A'\ if\ empty) \leftarrow$$

for every clause $A \leftarrow$, with variables $x_1, \ldots, x_k$, in P.

The most important properties of the vanilla interpreter, including its soundness and completeness, are given by theorem 3.1, corollary 3.2, and theorem 3.3 in [HL88].

We now discuss the ground representation, which is similar to the typed representation, except for the representation of variables. In the ground representation, a variable x in $\mathcal{L}$ is represented by a constant x' of type o in $\mathcal{L}'$ (in contrast to the typed representation, where x' is a variable). Otherwise, the ground representation is similar to the typed representation. The details are given in [HL88].

Next we discuss an interpreter useful for implementing co-routining, which was presented in [HL88] and is based on the ground representation. This interpreter captures the procedural semantics of normal programs and goals given by SLDNF-resolution and is a descendant of the "demo" interpreter first given in [BK82]. The main differences with [BK82] are that the interpreter handles negation and it (implicitly) obtains computed answers. The use of this kind of interpreter for implementing co-routining is discussed, for example, in [OH88].

The typed program **G**, based on the language $\mathcal{L}'$, consists of the following definition for *solve*,

$$\begin{aligned}&\forall_\mu x, y \ \ solve(x, y) \leftarrow \\ &\qquad succeed(x \ if \ x, y \ if \ empty)\end{aligned}$$

and the following definitions for *succeed*, *fail*, and *head_formula*, together with definitions for *conjunction_of_literals* and *derive* (and predicate symbols upon which *derive* and *conjunction_of_literals* depend).

$$\begin{aligned}&\forall_\mu x \ \ succeed(x \ if \ empty, x \ if \ empty) \leftarrow \\ &\qquad head\_formula(x)\end{aligned}$$

$$\begin{aligned}&\forall_\mu l, r, s, u, v, w, x, y, z \ \ succeed(x \ if \ y, z) \leftarrow \\ &\qquad select(y, l, positive(s), r, u) \wedge \\ &\qquad clause(w) \wedge \\ &\qquad derive(resultant(x, l, s, r), w, v) \wedge \\ &\qquad succeed(v, z)\end{aligned}$$

$$\begin{aligned}&\forall_\mu l, r, s, u, x, y, z \ \ succeed(x \ if \ y, z) \leftarrow \\ &\qquad select(y, l, negative(s), r, u) \wedge \\ &\qquad fail(empty \ if \ s) \wedge \\ &\qquad succeed(x \ if \ u, z)\end{aligned}$$

$$\begin{aligned}&\forall_\mu l, r, s, u, y \ \ fail(empty \ if \ y) \leftarrow \\ &\qquad select(y, l, positive(s), r, u) \wedge \\ &\qquad \forall_\mu w, z \ (fail(z) \leftarrow clause(w) \wedge \\ &\qquad\quad derive(resultant(empty, l, s, r), w, z))\end{aligned}$$

$$\begin{aligned}&\forall_\mu l, r, s, u, y \ \ fail(empty \ if \ y) \leftarrow \\ &\qquad select(y, l, negative(s), r, u) \wedge \\ &\qquad succeed(empty \ if \ s, empty \ if \ empty)\end{aligned}$$

$$\begin{aligned}&\forall_\mu l, r, s, u, y \ \ fail(empty \ if \ y) \leftarrow \\ &\qquad select(y, l, negative(s), r, u) \wedge \\ &\qquad fail(empty \ if \ s) \wedge \\ &\qquad fail(empty \ if \ u)\end{aligned}$$

$$head\_formula(empty) \leftarrow$$

$$\begin{aligned}&\forall_\mu x \ \ head\_formula(x) \leftarrow \\ &\qquad conjunction\_of\_literals(x)\end{aligned}$$

Given a normal program P, based on the underlying language $\mathcal{L}$, the program $\mathbf{G}_P$ consists of the above program **G**, together with definitions for *clause*, *select*, and predicate symbols upon which *select* depends.

We define *clause* as follows. There is a clause of the form

$$clause(A' \ if \ Q') \leftarrow$$

for every clause $A \leftarrow Q$ in P, and a clause of the form

$$clause(A' \ if \ empty) \leftarrow$$

for every clause $A \leftarrow$ in P.

Informally, the intended meanings of the predicate symbols *fail* and *succeed* in the program $\mathbf{G}_P$ are as follows. The predicate symbol *succeed* is intended to be true when the first argument represents a resultant[2] R, the second argument represents a resultant $Q \leftarrow$ and there is an SLDNF-refutation of $P \cup \{R\}$ with final resultant $Q \leftarrow$. The predicate symbol *fail* is intended to be true when the argument represents a normal goal $\leftarrow Q$ and $P \cup \{\leftarrow Q\}$ has a finitely failed SLDNF-tree.

The intention is that *derive* be a system predicate and that *select* be user defined. A version of *select* which (safely) implements the Prolog "leftmost literal" computation rule is given in [HL88].

Various basic properties of $\mathbf{G}_P$ are given in theorem 4.1, theorem 4.2, corollary 4.3, and theorem 4.4 in [HL88].

It is worth emphasising that the theories $\mathbf{V}_P$ and $\mathbf{G}_P$, and their corresponding intended interpretations, are very different. $\mathbf{V}_P$ is best regarded as a thin layer on top of the object program P and, as can be seen from theorem 3.1 in [HL88], it inherits its intended interpretation from the intended interpretation of P. In contrast, as theorem 4.2 in [HL88] shows, the declarative semantics of $\mathbf{G}_P$ is intimately related to the procedural semantics of P. In fact, as indicated above, the intended interpretation of $\mathbf{G}_P$ is given by SLDNF-resolution (and, in particular, is unrelated to the intended interpretation of P).

We now indicate how to give the definition of the meta-logical predicate symbol *var* and some other predicate symbols which the definition of *var* requires. Definitions of other meta-logical predicate symbols may be found in [HL88]. These definitions are all based on the ground representation. We assume that $\mathcal{L}$ contains the constants $a_1, \ldots, a_l$, and the function symbols $f_1, \ldots, f_m$ of arity $k_1, \ldots, k_m$, respectively.

We begin with the predicate symbol *constant* of type o in $\mathcal{L}'$, which is intended to be true when its argument is the representation of a constant in $\mathcal{L}$. The definition of *constant* is as follows.

$$\begin{aligned}&constant(a_1') \leftarrow \\ &\quad\vdots \\ &constant(a_l') \leftarrow\end{aligned}$$

[2]The definition of the concept of a resultant and its usefulness for meta-programming is explained in [HL88].

The definition of the predicate symbol *nonvar* of type o, which is intended to be true when its argument is the representation of a non-variable term in $\mathcal{L}$, is as follows.

$$\forall_o x \;\; nonvar(x) \leftarrow \\ \qquad constant(x)$$

$$\forall_o x_1, \ldots, x_{k_1} \;\; nonvar(f_1'(x_1, \ldots, x_{k_1})) \leftarrow \\ \vdots \\ \forall_o x_1, \ldots, x_{k_m} \;\; nonvar(f_m'(x_1, \ldots, x_{k_m})) \leftarrow$$

The definition of the predicate symbol *var* of type o, which is intended to be true when its argument is the representation of a variable in $\mathcal{L}$, is as follows.

$$\forall_o x \;\; var(x) \leftarrow \\ \qquad \neg nonvar(x)$$

This definition provides a satisfactory declarative and procedural semantics for *var* which avoids the kind of problem referred to in the first section.

It is possible to identify two different kinds of use of *var* in Prolog. The first is the "meta-level" use which can be understood as above. This use of *var* is exemplified by the unification algorithm given on page 152 of [SS86b]. The other use is the "control" use, which is exemplified by the program for *plus* given on page 147 of [SS86b]. This control use of *var* is confusing because it appears necessary to give some kind of declarative semantics to the corresponding *var* atoms appearing in the program. In fact, no such declarative semantics is necessary or even possible (as was pointed out in the first section). It would be preferable to disallow this use of *var* and replace it by explicit control annotations, which achieve the same effect.

There are at least two other ways of defining *var*. The most obvious way is to have a fact of the form $var(x') \leftarrow$, for each constant x' in $\mathcal{L}'$ which represents a variable x in $\mathcal{L}$. Since there are infinitely many variables in $\mathcal{L}$, this definition of *var* contains infinitely many facts. This method of defining *var* is the most direct, but it causes a problem with the completion of the definition. An alternative definition involves using certain ground terms in $\mathcal{L}'$ to represent the variables in $\mathcal{L}$. Let a be a distinguished constant of type o and s a distinguished unary function symbol of type $o \rightarrow o$ in $\mathcal{L}'$. Then we can represent the variables x_0, x_1, $x_2, \ldots$ in $\mathcal{L}$ by the ground terms a, $s(a)$, $s(s(a)), \ldots$ of type o in $\mathcal{L}'$. The definition of *var* is as follows.

$$var(a) \leftarrow \\ \forall_o x \;\; var(s(x)) \leftarrow var(x)$$

This definition avoids the previous problem with the completion. However, it leads to some complications with the definition of *nonvar*.

3 Research Problems

In this section, we discuss various research problems in meta-programming.

3.1 Typing

While Prolog systems have traditionally not had any typing, it is clear that meta-programming is greatly facilitated by having a type system. This requirement of a type system for Prolog is evident from many other points of view as well, so it is not necessary to give any further justification here.

What kind of type system does meta-programming require? At the very least, we require a many-sorted logic upon which the typed and ground representation schemes are based. Beyond this, the ability to define lattice structures on the sorts and also some form of polymorphism in the type system are often useful.

3.2 Representation

Prolog systems should make the typed and ground representations explicitly available to the programmer. In other words, some convenient notation should be available to refer to what we have denoted here by a', $f', \ldots$.

In this respect, the question of the relative merits of the typed representation versus the ground representation is an interesting one. Of those papers which study interpreters, the great majority employ the vanilla interpreter and its various enhancements, which are all based on the typed representation. There seem to be good reasons for the popularity of the vanilla interpreter. It is simple, short, easy to understand, and lends itself easily to various enhancements. However, we believe the actual situation is not nearly so straightforward as the papers expounding this viewpoint claim. The problem is that the example interpreters appearing in these papers are usually rather short and simple. In practice, interpreters based on these ideas are more complicated in that they use a variety of meta-logical predicates. As we have already pointed out, this is when the trouble starts, since interpreters based on the vanilla interpreter and using meta-logical predicates, such as *var*, do not have any declarative semantics. In other words, the apparent simplicity of this approach actually hides serious difficulties. The interpreters are more or less achieving the desired effect, but at the price of giving up the logic in Logic Programming. This is too high a price.

For these reasons, we argue that the current research effort being put into the vanilla interpreter and its enhancements is largely misplaced. It would be better to start with an interpreter based on the ground representation. A good candidate is the interpreter **G**, which also lends itself to various variations, refinements, and enhancements. This approach avoids the semantic problems with the meta-logical predicates.

3.3 Implementation

The major meta-programming requirements of a Logic Programming system are that

1. it should make available the typed and ground representations
2. it should make available the various meta-logical predicates (including those defined in [HL88]), and
3. it should achieve 1. and 2. efficiently.

Implementing 1. and 2. is easy, but doing it efficiently is a major research problem. The naive implementation of the ground representation leads to a substantial overhead in handling object level variables. How can we avoid this overhead? One possible approach is to implement the ground representation by the trick of representing object level variables by meta-level variables (instead of constants). Provided these meta-level variables are carefully handled, they will not cause any problems and, most of the time, will act like constants. However, it should then be possible to implement potentially expensive meta-logical predicates, such as *derive* and *unify*, by passing to the system's underlying unification algorithm. It should also be possible to exploit the Prolog system's low level representation of variables, rather than directly use the definition of *var* given above, but do this in such a way as to preserve the declarative and procedural semantics of *var* given by this definition.

Thus the implementation we have in mind is via a compilation process, whereby a meta-program is compiled (safely!) into a lower-level program similar to what Prolog programmers currently write. This compiled program can then be run directly and efficiently on current Prolog systems. From this perspective, one can regard current Prolog meta-programs as written in a low-level machine or assembler language and that we wish to allow programmers to program at a much higher level.

3.4 Arbitrary Programs and Goals

We have confined attention to providing representation schemes suitable for handling normal programs and goals. However, ultimately we want to be able represent (arbitrary) programs and goals [Llo87]. Programs are such that the body of a program statement can be an arbitrary first order formula. Similarly, the body of a goal can be an arbitrary first order formula. The increased expressiveness of programs and goals is useful for expert systems, deductive database systems, knowledge base systems, and general purpose programming.

The use of programs and goals means that we must have some way of representing formulas with quantifiers. For the ground representation, this can be handled by what appears to be a very straightforward extension of the framework and results of [HL88]. (See [Llo87] for an application of this extension to a declarative error diagnoser.) However, for the typed representation, the representation of formulas with quantifiers takes us outside first order logic into some kind of higher order logic with λ-terms [MN86]. This path is also worth following as a move to the greater expressiveness of higher order logics will probably be generally beneficial, not just for meta-programming. Other work closely related to this issue, but using first order logic, is contained in [SRP*88].

3.5 Systematic Construction of Meta-Programs

One can divide the task of building expert systems into two phases: the knowledge elicitation phase, in which the expert knowledge is somehow gathered, and the software engineering phase, in which the design and construction of the software component of the expert system is carried out. Knowledge elicitation is currently a black art and is likely to remain so for a long time. However, in contrast to the current, largely *ad hoc* approaches in the AI community, the software engineering phase ought to be susceptible to systematic construction techniques.

Meta-programming techniques offer the possibility of building a theory of expert system shells. Indeed, it is possible that a calculus of operations on meta-programs could be the basis of such a theory. One can easily imagine an expert system toolbox containing provably correct components for performing certain tasks. For example, the toolbox could contain a variety of interpreters each with a single function, such as constructing a proof tree, performing some kind of uncertain reasoning, and so on (the *flavours* of [SB86]), together with useful utilities such as partial evaluators. Suppose now that a programmer wanted to write an interpreter combining several of the functions. It would be convenient if it were possible to write (or, better still, choose from the toolbox) separate interpreters, each with one of the functions, and then have these interpreters automatically combined into the required interpreter, which would then be guaranteed to be correct. This process would also require automatic partial evaluation of the combined interpreter with respect to the object theory to produce an efficient result.

Currently, it is only possible to combine simple interpreters which have a strong structural similarity [SL88]. However, these ideas are certainly very promising and deserve much greater attention, both from the theoretical and practical perspective, than they have had so far.

3.6 Variants of the Interpreter G

The interpreter **G** was designed to implement the definitions of SLDNF-refutations and finitely failed SLDNF-trees. Consider the following variant of this interpreter.

$$\forall_\mu x \;\; succeed(x \; if \; empty, x \; if \; empty) \leftarrow \\ \quad head\_formula(x)$$

$$\forall_\mu l, r, s, u, v, w, x, y, z \;\; succeed(x \; if \; y, z) \leftarrow \\ \quad select(y, l, positive(s), r, u) \wedge \\ \quad clause(w) \wedge \\ \quad derive(resultant(x, l, s, r), w, v) \wedge \\ \quad succeed(v, z)$$

$$\forall_\mu l, r, s, u, x, y, z \;\; succeed(x \; if \; y, z) \leftarrow \\ \quad select(y, l, negative(s), r, u) \wedge \\ \quad \neg succeed(empty \; if \; s, empty \; if \; empty) \wedge \\ \quad succeed(x \; if \; u, z)$$

The difference between **G** and the variant is that **G** handles negation with the predicate *fail* while the variant handles negation with $\neg succeed$. In [HL88], it was conjectured that this variant interpreter captures the procedural semantics of stratified normal programs and normal goals given by SLS-resolution [Prz87]. It would be interesting to prove analogous results for this variant to those obtained in [HL88] for **G**.

3.7 Partial Evaluation

Partial evaluation is another topic which has recently attracted great interest. Its application to meta-programming is now well known. (See, for example, [CFL*88], [LS87a], [LS87b], [Neu86], [OH88], [SS86a], [SB86], [TF86] and [Ven84].) For the typed representation, the usefulness of partial evaluation is clear from the proof of theorem 3.3 in [HL88], where partial evaluation is used to remove a layer of interpretation by eliminating the calls to *clause*. In the case of the vanilla interpreter, (a subprogram of) the partially evaluated version of $\mathbf{V}_P$ is actually "isomorphic" to P.

As was pointed out in [Owe88], most papers studying partial evaluation of interpreters have considered only simple interpreters, usually modest extensions of the vanilla interpreter. However, [Owe88] presents four more complicated interpreters for which partial evaluation seems to be much more difficult. For example, the partial evaluation of these interpreters seems to require the introduction of new predicates and their definition, and the use of the fold operation. In this regard, the theoretical foundations of partial evaluation given in [LS87b] need to be extended to cover the introduction of new predicates and folding.

The interpreters in [Owe88], if presented formally, would be based on the ground representation. Thus they raise the general issue of the partial evaluation of meta-programs based on the ground representation, which is is not at all clear at the moment and needs further investigation.

3.8 Explanation Facilities

Explanation facilities are an important component of expert systems. However, only a few papers have been concerned with explaining answers which involve reasoning with negation. Two recent papers which do handle negation are [BH88] and [YS88]. However, ultimately we want to be able to give explanations in systems where the knowledge base is an (arbitrary) program. Since SLDNF-resolution for (arbitrary) programs is implemented by first transforming the program to normal form [Llo87], the problem is to relate the explanation extracted from the SLDNF-refutation or finitely failed SLDNF-tree back to the original program as written by the programmer.

3.9 Dynamic Knowledge Bases

The semantics of assert and retract is a continuing problem in Logic Programming. A recent paper on this topic is [Sub88], in which a semantics for the MetaProlog system [BW85] is given. However, this problem deserves much greater attention from theoreticians.

Closely related to this is the general issue of updating knowledge bases, which is also poorly understood. A general setting for the update problem is as follows. Given an (arbitrary) program, satisfying some integrity constraints, and a closed first order formula, which is (resp., is not) a logical consequence of the completion of the program, find a way to change the program so that the formula is no longer (resp., is) a logical consequence of the completion of the modified program and so that the modified program also satisfies the integrity constraints. This problem can be regarded as a generalisation of the well known view update problem from relational databases.

Special cases of this problem can easily be solved. For example, if the program is definite and we want to delete an atom, we can run the atom as a goal and then systematically look for ways of cutting the success branches of the search tree by deleting facts in the program. Those collections of deletions of facts in the program which also satisfy the integrity constraints give ways of achieving the deletion of the atom. Once negation is introduced the problem becomes much more subtle, but this kind of approach can be extended.

3.10 Program Transformation

Program transformation is an area of Logic Programming which hasn't attracted sufficient interest considering its potential importance and practical application. There have only been a modest number of papers on this topic which have usually restricted attention to definite programs and are usually only concerned with the restricted semantics of the least Herbrand model. For the systems languages, such as Parlog, Concurrent Prolog and GHC, this may be an appropriate setting. However, for the applications languages, negation is vital and cannot be ignored. What is required here is a systematic study of program transformation in the setting of (arbitrary) programs. The major theoretical questions are then of the form: Does a particular transformation (or series of transformations) produce a transformed program with the same correct answers (or computed answers or completion) as the original program. The major practical problem is to provide sufficient control to the transformer so as to obtain the desired transformed program with the least possible intervention by the programmer.

3.11 Abstract Interpretation

Abstract interpretation is another area of great promise. However, almost all papers have been confined to the setting of definite programs. What is required here is a systematic study of abstract interpretation in the setting of normal programs. It may be necessary to restrict attention to stratified, or more generally, call-consistent, normal programs, for which the mapping T_P, and presumably also its abstract counterparts, have a useful monotonicity property (proposition 17.3 of [Llo87]).

4 Beyond Prolog

In this paper, we have examined the deficiencies in the meta-programming facilities provided by Prolog. In a similar way, we could have given an analysis of Prolog's other deficiencies, such as lack of occur check, unsafe negation, undisciplined use of cut, assert, retract, and so on. Taken together, these deficiencies provide convincing evidence that the Logic Programming community must make a serious effort now to build a successor to Prolog.

The successor to Prolog that we have in mind will have much the same expressive power as Prolog, but will have significant improvements in the declarative understanding of programs. Fortunately, all the research done so far on Prolog implementation will be applicable to the new language. But we have to face the fact that the improvements in the language will probably involve some overheads compared to Prolog. For example, it is likely that the implementation of appropriate meta-programming facilities, along the lines of those described here, will incur some overhead.

Undoubtably, Prolog has served a useful purpose. It has demonstrated the viability of logic as a programming language, it has motivated numerous research projects concerned with the next generation of computing systems, and it has attracted large numbers of able people to the field. The language is currently being standardised, which will ensure that, warts and all, it will be with us in its present form well into the next century. However, while there are sound reasons for standardising Prolog, there is no excuse for thinking that it is a close approximation to the best that can be achieved. On the contrary, Prolog is best regarded as the Fortran of Logic Programming languages. It will take a great deal of hard work and commitment from the Logic Programming community to achieve the goal of creating really satisfactory Logic Programming languages.

References

[BH88] A. Bruffaerts and E. Henin. Proof trees for negation as failure: yet another prolog meta-interpreter. In J.W. Lloyd, editor, *Workshop on Meta-Programming in Logic Programming*, pages 133–146, Bristol, June 1988.

[BK82] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 153–172, Academic Press, London, 1982.

[Bow85] K.A. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, 3(3):359–383, 1985.

[BW85] K.A. Bowen and T. Weinberg. A meta-level extension of prolog. In *IEEE Symposium on Logic Programming*, pages 669–675, Boston, 1985.

[CFL*88] P. Coscia, P. Franceschi, G. Levi, G. Sardu, and L. Torre. Object level reflection of inference rules by partial evaluation. In P. Maes and D. Nardi, editors, *Meta Level Architectures and Reflection*, North-Holland, 1988.

[Esh86] K. Eshghi. *Meta-Language in Logic Programming.* PhD thesis, Department of Computing, Imperial College, 1986.

[HL88] P.M. Hill and J.W. Lloyd. *Analysis of Meta-Programs.* Technical Report CS-88-08, Department of Computer Science, University of Bristol, 1988.

[Kow79] R.A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

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

[Llo88] J. W. Lloyd, editor. *Workshop on Meta-Programming in Logic Programming,* Bristol, June 1988.

[LS87a] G. Levi and G. Sardu. Partial evaluation of metaprograms in a "multiple worlds" logic language. In D. Bjorner, A.P. Ershov, and N.D. Jones, editors, *Workshop on Partial Evaluation and Mixed Computation,* pages 213–223, Gl. Avernaes, Denmark, October 1987.

[LS87b] J.W. Lloyd and J.C. Shepherdson. *Partial Evaluation in Logic Programming.* Technical Report CS-87-09, Department of Computer Science, University of Bristol, 1987.

[MKK*83] T. Miyachi, S. Kunifuji, H. Kitami, K. Furukawa, A. Takeuchi, and H. Yokota. *A Knowledge Assimilation Method for Logic Databases.* Technical Report TR-025, ICOT, 1983.

[MN86] D.A. Miller and G. Nadathur. *Higher-Order Logic Programming.* Technical Report MS-CIS-86-17, Department of Computer and Information Science, University of Pennsylvania, 1986.

[Neu86] G. Neumann. *Meta-Interpreter Directed Compilation of Logic Programs into Prolog.* Technical Report RC 12113, IBM T.J. Watson Research Center, 1986.

[OH88] S. Owen and R. Hull. *The Use of Explicit Interpretation to Control Reasoning about Protein Toplogy.* Technical Memo, Hewlett-Packard Bristol Research Centre, 1988.

[Owe88] S. Owen. Issues in the partial evaluation of meta-interpreters. In J.W. Lloyd, editor, *Workshop on Meta-Programming in Logic Programming,* pages 241–254, Bristol, June 1988.

[Prz87] T. Przymusinski. On the declarative and procedural semantics of logic programming. 1987. Unpublished manuscript.

[SB86] L.S. Sterling and R.D. Beer. *Meta-Interpreters for Expert System Construction.* Technical Report TR 86-122, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1986.

[SL88] L.S. Sterling and A. Lakhotia. *Composing Prolog Meta Interpreters.* Technical Report TR 88-05, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1988.

[SRP*88] J. Staples, P. Robinson, R. Paterson, R. Hagen, A. Craddock, and P. Wallis. Qu-prolog: an extended prolog for meta level programming. In J.W. Lloyd, editor, *Workshop on Meta-Programming in Logic Programming,* pages 319–332, Bristol, June 1988.

[SS86a] S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Information Processing 86,* pages 271–278, North Holland, Dublin, 1986.

[SS86b] L.S. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Sub88] V.S. Subrahmanian. Foundations of metalogic programming. In J. W. Lloyd, editor, *Workshop on Meta-Programming in Logic Programming,* pages 53–66, Bristol, June 1988.

[TF86] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86,* pages 415–420, North Holland, Dublin, 1986.

[Ven84] R. Venken. A prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimization. In *ECAI-84: Advances in Artificial Intelligence,* pages 91–100, North-Holland, Pisa, 1984.

[YS88] L.U. Yalçinalp and L.S. Sterling. An integrated interpreter for explaining prolog's successes and failures. In J.W. Lloyd, editor, *Workshop on Meta-Programming in Logic Programming,* pages 147–160, Bristol, June 1988.