Abstract

In this review of metaprogramming in logic we pay equal attention to theoretical and practical issues: the contents range from mathematical and logical preliminaries to implementation and applications in, e.g., software engineering and knowledge representation. The area is one in rapid development but we have emphasized such issues that are likely to be important for future metaprogramming languages and methodologies.
1 Introduction

The term ‘metaprogramming’ relates to ‘programming’ as ‘metalanguage’ relates to ‘language’ and ‘metalogic’ to ‘logic’: programming where the data represent programs. It should be no surprise that metaprogramming with logic programming languages takes advantage of many results from metalogic.

In the most general interpretation we would say that ‘metaprogramming’ refers to any kind of computer programming where the input or output represents programs. We will refer to a program of this kind as a metaprogram and to its data as object programs. Analogously, we will call the languages in which these programs are written a metaprogramming language and an object programming language, respectively.

There are several applications where metaprogramming is employed already today, not always consciously by the programmer. An obvious existing class of applications encompasses compilers, program transformation and analysis tools, and other software engineering programs (popularly known as CASE tools). Interpreters may also fall within this class but borders on another major application area: knowledge representation and reasoning. Metaprogramming enables writing of programs that treat other knowledge, coded in the form of runnable programs, as data.

For logic and metalogic we will essentially follow the terminology, if not the syntax, of Schönf"{u}ld [79]. However, our languages will be those of normal programs and our theories will always have SLDNF-resolution as an inference rule.

For the benefit of readers not familiar with logic programming we conclude this introduction with a brief summary of some basic concepts of logic programs. The reader is directed to Lloyd’s textbook on foundations of logic programming [59] for a more thorough introduction and precise definitions.

By a term we will understand either a constant symbol, a variable symbol or a function symbol followed by a sequence of terms. By an atomic sentence (or simply an atom) we mean either a propositional constant symbol or a predicate symbol followed by a sequence of terms. A literal is either an atomic sentence or a negated atomic sentence.

A program clause is a sentence $\forall \bar{\exists} (A \leftarrow L_1 \land \cdots \land L_n)$, $n \geq 0$, where $A$ is an atomic sentence, every $L_i$, $1 \leq i \leq n$, is a literal, and $\bar{\exists}$ is the set of all variables occurring in $\{A, L_1, \ldots, L_n\}$. We call $A$ the head and $L_1 \land \cdots \land L_n$ the body of the program clause (note the similarity between a program clause and a procedure abstraction in a conventional programming language). A normal program is a finite set of program clauses. A goal clause is a sentence $\exists \bar{\exists} (\leftarrow L_1 \land \cdots \land L_n)$, $n \geq 0$, every $L_i$, $1 \leq i \leq k$ is a literal, and $\bar{\exists}$ is the set of all variables occurring in $\{L_1, \ldots, L_n\}$. It can be understood as an existentially closed implication having a trivially false
consequent.

A substitution is a mapping between expressions that consistently replaces all occurrences of some variables with other expressions. Application of a substitution \( \theta \) to an expression \( E \) is commonly written \( E\theta \).

A substitution \( \theta \) is a unifier for a set of atoms \( \{A_1, \ldots, A_k\} \) if \( \{A_1\theta, \ldots, A_k\theta\} \) is singleton, i.e., \( A_1\theta = \cdots = A_k\theta \). A unifier \( \sigma \) is a most general unifier (mgu) for a set \( S \) of atoms if for any unifier \( \theta \) of \( S \), there exists a substitution \( \lambda \) such that \( \theta = \sigma \circ \lambda \), where \( \circ \) is functional composition. Not all sets of atoms have unifiers.

SLD-resolution is a proof procedure that can be thought of as repeated rewriting of a goal clause, each time using a program clause. An mgu is computed for the selected atom\(^1\) of the goal clause and the head of the program clause. If this succeeds, the selected atom is replaced in the goal clause by the literals of the body of the program clause, and the mgu is applied to the resulting goal clause. Procedurally, application of the mgu corresponds to (but generalizes) the passing of parameters in a conventional programming language, while replacement of the selected atom corresponds to a subroutine call. A proof succeeds (and execution thus terminates with success) if a goal clause with no literals is reached. A proof fails (execution terminating with failure) if, at some step, the selected atom cannot be rewritten using any program clause of the program.

SLDNF-resolution adds an inference step also for selected negated atoms that has no direct analogue in execution of conventional programming languages. If the selected literal is \( \neg B \) (containing the variables \( \overline{w} \)) and rewriting a goal clause \( \exists \overline{w}(\neg B) \) has terminated with failure, then \( \neg B \) can simply be removed from the goal clause.

2 Representation

Following the seminal paper by Bowen & Kowalski [9, 52], metalogic programming has mostly been concerned with provability, i.e., whether an expression \( Q \) can be obtained through finitely many applications of one or more inference rules, beginning with a finite set of expressions \( \{P_1, \ldots, P_k\} \) for some \( k, k \geq 0 \).

We choose to follow a slightly different course of discussion in terms of theoremhood, i.e., whether an expression \( Q \) is a theorem of a theory \( T \), i.e., whether \( Q \) belongs to the smallest set of expressions that contains the nonlogical axioms of \( T \) and is closed under inference.\(^2\)

\(^1\)The selection function could be any total function that selects one literal from a set. The selection function used in the Prolog logic programming language [18] always picks the leftmost conjunct of the goal clause.

\(^2\)The axioms of a theory can be divided into logical axioms that "come with the language", e.g., equality axioms, and nonlogical axioms that are particular for the theory under consideration.
If we let $T[P_1, \ldots, P_k]$ denote the theory obtained from $T$ by adding $P_1, \ldots, P_k$ as nonlogical axioms, then (according to the deduction theorem for first-order theories) the notion of proving $Q$ from $\{P_1, \ldots, P_k\}$ in $T$ is equivalent to the notion of $Q$ being a theorem of $T[P_1, \ldots, P_k]$. Thus we do not lose any expressivity by this change of focus, but we find that it makes our exposition of the subject more elegant and that it conforms well with recent advances in metalogic programming.

2.1 Separated Languages

Bowen & Kowalski gave a first definition of what it means for a metalanguage to represent an object language.\(^3\) We say that a metalanguage $\mathcal{L}_M$ represents an object language $\mathcal{L}_O$, if the following proposition is true.

For every expression $E$ in $\mathcal{L}_O$, there exists at least one ground (i.e., variable-free) term\(^4\) $E'$ in $\mathcal{L}_M$ that represents $E$ (it is also common to say that $E'$ is a name in $\mathcal{L}_M$ for $E$), such that if two expressions $E_1$ and $E_2$ in $\mathcal{L}_O$ are distinct, then so are $E'_1$ and $E'_2$.

We will also say that a set $S'$ of terms in $\mathcal{L}_M$ represents a set $S$ of expressions in $\mathcal{L}_O$, if there is a one-to-one correspondence between elements of $S'$ and $S$ under representation.

We say that a metatheory $\mathcal{T}_M$ (whose language is $\mathcal{L}_M$) represents an object theory $\mathcal{T}_O$ (whose language is $\mathcal{L}_O$), if there exists in $\mathcal{L}_M$ a predicate symbol, which we will here assume to be $\text{Demo}_{\mathcal{T}_O}$, such that the following proposition is true for any sentence $P$ in $\mathcal{L}_O$ and term $P'$ in $\mathcal{L}_M$ that represents $P$.

$$\vdash_{\mathcal{T}_O} P \quad \text{if and only if} \quad \vdash_{\mathcal{T}_M} \text{Demo}_{\mathcal{T}_O}(P').$$

This last proposition is known as a reflection principle, and was employed by Weyhrauch in his FOL system [89] but can be traced back to Feferman [26].

Note that we cannot require that $\vdash_{\mathcal{T}_M} \neg \text{Demo}_{\mathcal{T}_O}(P')$ whenever $\not\vdash_{\mathcal{T}_O} P$, or vice versa. Provided that $\mathcal{T}_O$ is consistent and contains the nonlogical axioms of arithmetic, it follows from Church’s theorem [17] that there can be no finite representation, as nonlogical axioms of a theory $\mathcal{T}_M$, of its nonprovability (alternatively Gödel’s incompleteness theorem [33] can be used to contradict a complete axiomatization of nonprovability in $\mathcal{T}_O$). (The

---

\(^3\)They called such a representation an amalgamation of the metalanguage and the object language, but this term has since come to denote a stronger concept as discussed in Sect. 2.3.

\(^4\)We could weaken this restriction to require only that the representing term is closed, as is common in mathematical logic [80], but the terminology in the field has long distinguished mainly between ground and nonground representations.
consequence of this restriction for metaprogramming was observed by Bowen 
& Kowalski [9].

However, as Konolige points out [50], being able to show that a sen-
tence is not a consequence of some knowledge is essential for many forms of
introspective reasoning, so completely abandoning the negation of Demo is
probably going too far. For example, if $T_O$ is weak enough to be decidable,
then the negation of $Demo_{T_O}$ can be axiomatized. Eshghi discusses other
approximations of nonprovability [25].

Given that a language $L_M$ represents a language $L_O$ it can make compu-
tational sense to employ the reflection principle above as additional inference
rules

\[
\frac{
\vdash_M Demo_{T_O}(P')
}{\vdash_O P}
\quad \text{and} \quad
\frac{
\vdash_O P
}{\vdash_M Demo_{T_O}(P')}
\]

in order to make proofs shorter. We call these rules meta-to-object and
object-to-meta reflection, respectively.\(^5\)

Alternatively we may start with a pair of theories $T_M$ and $T_O$ where
the language of $T_M$ represents the language of $T_O$, but where $T_M$ does
not represent $T_O$. By adding the inference rules above to $T_M$ and $T_O$, we
obtain theories $T'_M$ and $T'_O$ such that $T'_M$ represents $T'_O$. (Note that these
inference rules are improper in that they mix the languages $L_M$ and $L_O$.
We might resolve this problem by representing all involved languages in a
single ‘superlanguage’.)

It is conceivable that a theory $T_M$ represents a collection of $l$ theories
$T_{O_1}, \ldots, T_{O_l}$, $l \geq 0$. Provided that the language of $T_M$ contains distinct
expressions $O'_1, \ldots, O'_l$ that denote the theories $T_{O_1}, \ldots, T_{O_l}$, we can let
$Demo$ from now on be a binary predicate symbol such that $\vdash_M Demo(\neg_O p)\)
if and only if $\vdash_O P$, where $1 \leq i \leq l$ and $P'$ represents $P$.

We may ask what has been gained by representing the language $L_O$ in
$L_M$, as proving a theorem $Demo(O'_i, P')$ in $T_M$ instead of proving $P$
directly in $T_O$ brings nothing new. However, it is possible to express and
solve sentences concerning $T_O$ in $L_M$ that cannot even be expressed directly
in $L_O$. One example is $Demo(O'_1, p)$ (where $p$ is a variable in $L_M$); every
answer to this query equates $p$ with a representation of a sentence $P$ in $L_O$
that is a theorem in $T_O$. Another example is $Demo(O'_1[u], P') \land \text{Acceptable}(u)$
(where $u$ is a variable in $L_M$); answers to this query equate $u$ with some
set of representations of sentences $U$ in $L_O$, such that the sentence $P$ is
a theorem in $O$ extended with $U$ and that the new sentences fulfill some
acceptance criteria (e.g., some integrity constraints). Such a query performs
a form of abduction, not as an inference mechanism of $T_O$, but through a
deduction at the metalevel [55].

\(^5\)These rules have often been called downward and upward reflection. Unfortunately
some authors have swapped the meaning of the names and we prefer to abandon them.
2.2 Languages that Represent Themselves

An interesting special case occurs when \( L_M \) and \( L_O \) are the same language. It means that the language contains a representation of its own expressions. An important property of such a language is that it is possible to employ arbitrarily many metalevels. We will call such languages self-representable. We consider such languages particularly interesting because we believe that future applications of metaprogramming will often use metameta and higher levels. Our views in the rest of this paper should be read with this consideration in mind.

For example, the Gödel language (see Sect. 5.4) provides such a representation of itself (although through an abstract datatype) and there is also a module that contains a representation of Gödel provability. Another property of a self-representable language is that self-referential sentences can be constructed [9]. This is not harmful in itself, but a theory with such a language can express paradoxes.

The concept of self-reference has been studied quite extensively, e.g., by Perlis [70, 71]. There is an overview of self-reference and incompleteness in logic by Smoryński [80].

2.3 Amalgamation: Theories that Represent Themselves

Given a self-representable language \( L_A \), we may go one step further and develop a theory \( T_A \) that represents itself, i.e., having the property that \( \text{Demo}_{T_A}(P') \) is a theorem of \( T_A \) if and only if \( P \) is a theorem of \( T_A \), where \( P' \) represents \( P \) in \( L_A \).

Clearly, such an amalgamated theory that contains ‘interesting’ theorems about its own theorems may well be inconsistent. Still, autoepistemic theories are interesting for formalizing agents that may reflect upon their knowledge or deductive capabilities [50, 66].

One example of a metalogic programming language for expressing an amalgamated theory is Reflective Prolog (see Sect. 5.3). As a further example of an amalgamated theory, see the representation by Kim & Kowalski [46] of the Three Wise Men problem (Sect. 6.3).

2.4 Theories as Data Structures

We may think of a logic program or a knowledge base as a theory (with the program clauses being the axioms). Many sensible operations on programs and knowledge bases can be expressed in terms of operations on theories.

Given that the language and inference rules of a collection of theories is fixed, we may characterize each theory by its set of nonlogical axioms. Brogi, Mancarella, Pedreschi & Turini have studied operations on theories and propose an algebra on theories with operations such as union and difference [16]. This work is representative of a common approach (employed, e.g., in 'LOG [16]): to represent a theory by some data structure (e.g., a
list), which contains as parts a representation of every nonlogical axiom of the theory.

Kowalski has proposed a quite different approach [55] where each theory is instead represented by a ground term (that may even be a constant) for which the correspondence between the representation of a theory and its theorems is given separately. This idea has the advantage that reasoning about theories becomes more compact and, in a sense, more abstract because we may talk about a theory without reference to its actual axioms or theorems. This seems particularly useful when representing the beliefs of a number of agents, which may involve beliefs about other agents’ beliefs.

Midway between these approaches we find a proposal, also by Kowalski [53], that is used in MetaProlog [7], where theories are denoted by constants but where “virtual” theories can be constructed by combining other theories. Barklund & Hamfelt compare the two methods for expressing a typical problem in representation of legal knowledge [6].

Note that the naming of theories in Sect. 2.1 is neutral concerning the choice of representation of theories.

3 Representation of the Object Language

In this section we will discuss representations of expressions in an object programming language. Because we will discuss implementation-oriented topics we will often write ‘names of expressions’ rather than ‘representations of expressions’, in order to avoid confusion between the expressions in a metalanguage that represent object language expressions and the internal representations of expressions (in either language).

Recall that the definition of language representation above requires that any expression \( E' \) in \( L_M \), representing an expression \( E \) in \( L_O \), must be ground. A “non-ground representation”, containing variables, is thus a contradiction in terms. However, such naming conventions have often been applied in Prolog [18] metaprograms and merit further investigation. Below we will therefore use the term “non-ground representation” when discussing certain improper representations; their advantages and disadvantages.

3.1 Ground Representations

In a metaprogramming language, several pragmatic issues must influence the choice of representation of an object language. It is dictated that a term \( E' \), naming \( E \), must be ground, but it places no other restrictions on the structure of \( E' \). Following Tarski [86] we shall consider flat quotation-mark names of expressions (strings), as well as structural-descriptive names (compound nested terms).

Flat names One natural representation uses strings, e.g., finite sequences of characters. Strings are obviously variable-free and are rich enough to represent any object language. It is sufficient to encode the alphabet of
$L_O$ as distinguishable strings in $L_M$, a name of an $L_O$-expression is then obtained as the concatenation of the encodings of the symbols making up the expression. For example, if the syntax of a string in $L_M$ is a sequence of characters surrounded by double quotes, then one name of an $L_O$-sentence $\exists x(P(x) \land Q(x) \to R(x) \lor S(x))$ could be

"exists x(P(x) & Q(x) \rightarrow R(x) ; S(x))".

Metavariables (ranging over strings) can occur in names through concatenation. Suppose that the operator $++$ denotes string concatenation, then the expression

"exists x(P(x) & Q(x) ; " ++ y ++ " & S(x))"

is a partially instantiated name of a logical expression; if the name of a logical expression $\phi$ were to be substituted for the metavariable $y$, then the whole expression above would be a name of $\exists x(P(x) \land Q(x) \lor \phi \land S(x))$. In order to be practically useful, it seems necessary for the metalanguage to be equipped with string unification that handles the string concatenation operator.

As is well known, strings may be very expensive data structures. For example, a unification algorithm expressed in $L_M$ that manipulates string representations of expressions in $L_O$ (e.g., Robinson's original unification algorithm [75]) may require exponential time as well as exponential space. Such metaprograms would usually be too inefficient for practical use.

On the other hand, a string representation has the advantage that not only well-formed object language expressions but also fragments of well-formed expressions can be represented and ranged over by metavariables.

It is also worth mentioning that files can be seen as a form of persistent strings, although with limited possibilities for manipulation. Therefore what has been argued above holds for such representations as well.

**Structured names** For efficiency reasons implementations of logic programming languages employ pointer-based implementations of terms, where distinct variable symbols have been replaced by, e.g., references to distinct memory locations. It is desirable to use such implementations also for names of terms. There are also other reasons for investigating structured representations of expressions; the normal way to express computation in logic programs is through manipulation of compound expressions; unification is usually defined for the ordinary terms of predicate calculus [59, 75]. As efficiency is a significant reason for considering structured representations, we will approach this issue by considering the possible underlying implementation of such names.

It should be noted that pointer-based implementations of terms can be used in logic programming languages only because the operations that
can be applied to terms in these languages are quite restricted. In particular, only proper expressions and sequences of proper expressions are ever present in proofs and substitutions. Similar restrictions on the use of representations of terms in the metaprogramming language are required for pointer-based implementation of names of expressions to be feasible. Experience so far indicates that metaprograms tend to manipulate only names of proper expressions and names of sequences of proper expressions, with two exceptions.

1. Metavariables of appropriate type may be present in place of proper subexpressions.

2. Function and predicate symbols having a nonzero arity are usually not proper expressions but their names are often considered to be proper expressions.

The first exception is an important one, because it means that metalanguage variables and names of variables must be implemented in such a way that they can always be distinguished from each other. In a proper implementation this is trivial because names of variables were required above to be ground, but it points out one of the potential problems with improper nonground representations as discussed in the next section.

A similar problem might also appear in implementations that allow metalevel and object level computations to be mixed. We mentioned above that efficient implementations of logic programming languages do without variable symbols altogether. It may therefore be necessary to devise ways of distinguishing ‘anonymous’ references to variables in the metalanguage from similar ‘anonymous’ references to variables in the object language. We let a structural-descriptive name of an expression $E$ be an expression $E'$ such that $E'$ contains as proper subexpressions the names of all proper subexpressions of $E$. The definition of representation requires that distinct expressions have distinct names. It is therefore convenient, although not strictly necessary, to let the function symbol of $E'$ depend on whether $E$ is a variable, a symbol or a compound expression. The choice of names for variables in $E$ is particularly difficult [4]. If the variables in $E$ have a scope, then that scope must somehow be reflected in $E'$. The fact that Horn clauses are usually presented without explicit quantifiers appears to be the source of much confusion about names of variables. We know of two useful solutions: either quantifiers are made explicit, as in MetaProlog [7], and the language discussed by Kowalski [55], or some construct for delimiting scope in names is introduced. For example, one might decide that the second argument of $Demo$ is to be considered implicitly universally quantified; i.e., if the second argument is a name of an expression $E$, then it is actually a name of the universally closed expression $\forall E$.
Hill & Lloyd discuss a *typed* structural-descriptive representation of expressions [38]. Such a representation solves the problem of distinguishing between object level variables and metalevel variables but does not address the question of variable scope in names.

In the Gödel language, names of Gödel expressions form an abstract datatype, but it can be seen as a structural-descriptive representation because the predicates with which names can be manipulated denote relations between names of expressions and names of their subexpressions. The language also provides predicates that relate such structured names with a flat string representation.

It should be valuable for both flat and structured names if the representation of expressions is *compositional*, i.e., if the representation of an expression is made up from the representations of its subexpressions. Such representations have the advantage that one might compute part of the representation of an expression even if some subexpressions are not known at that time. For inference mechanisms that transfer information between metalevels, a compositional representation should allow more opportunities to carry on computation even if some representations of expressions cannot be fully determined.

In most or all metaprogramming languages, the language representation is *fixed*. This is not very flexible and renders the language useless for metaprogramming over representations of other languages or extended languages. It therefore makes sense to make language representations *definable* and van Harmelen has discussed which properties that such definable language representations must satisfy [87].

### 3.2 Nonground Representation

In the context of programming in the logic programming language Prolog, a certain form of metaprogramming has been employed for a long time. A sentence is represented by a term having the same syntax (Prolog allows such overloading of symbols) and object level variables are represented by metalevel variables. Given our definition of representation above, the expression ‘nonground representation’ is a contradiction in terms, but we will nevertheless employ it here.

The use of such a nonground representation\(^6\) of course only makes sense when the object language is the same as the metalanguage (as in the case in Prolog), or is sufficiently similar. In Prolog it is also common to make use of the fact that there is only a single form of expressions that, depending on the context, is read as either a predicate definition, a truth-valued expression (‘goal’) or a term.\(^7\) There is thus an obvious choice of a term to represent a

\(^6\)Given our definition of representation above, this expression is a contradiction in terms, but we will nevertheless employ it here.

\(^7\)This syntactic ambiguity appears to have caused much misunderstanding among those
goal: the term having the same syntax as the goal expression, variables and all.

Prolog contains the unary predicate `Call` that approximates a `Demo` predicate where the implicit theory is the database of program clauses; this database can be destructively increased or decreased by the `predicates` `Assert` and `Retract`. The `Call`, `Assert` and `Retract` predicates all employ the nonground representation.

One of the most popular uses of the nonground representation in Prolog is for writing interpreters, which is also the subject of the next section.

4 Interpreters

In the context of computational logic, an interpreter is the representation of the provability relation for a language $\mathcal{L}_O$ as an executable program in a metalinguage $\mathcal{L}_M$. Typically the interpreter manipulates representations of $\mathcal{L}_O$ expressions, for example, by applying a resolution inference rule to representations of clauses, yielding the representations of resolvents or substitutions (unifiers). It is often the case that the interpreter is logically incomplete with respect to the actual provability relation, e.g., Prolog interpreters are incomplete due to the depth-first search. Interpreters for a nontrivial logic programming languages that are both complete and efficient may turn out to be unobtainable, although there are interesting developments in this direction [78].

If $\mathcal{L}_O$ is ‘similar’ to $\mathcal{L}_M$ (e.g., when $\mathcal{L}_O = \mathcal{L}_M$), then we may call the interpreter a metainterpreter. From a practical point of view, interpreters (and in particular metainterpreters) provide a flexible way of implementing languages. A prototype can often be constructed very rapidly, sometimes in a few hours. This is particularly the case for metainterpreters in Prolog.

**Interpreters in Prolog.** It is common to write metainterpreters in Prolog using the nonground representation introduced previously. The significant advantage of the nonground representation is that the interpreter can simulate SLD-resolution of the object language using the unification and backtracking of the metalanguage (i.e., in the Prolog language processor). The substitutions are then never directly available to the interpreter but are incorporated in the answer substitution for the query to the interpreter.

These interpreters are often intended to be extensions of Prolog, or to provide additional information in answers, such as a proof tree for successful attempting to learn Prolog and even in the scientific community about the status of Prolog expressions. However, there are attempts to formalize and exploit this ambiguity [41].

Our vague definition of this term is chosen to cover what appears to be the most widely accepted meaning. However, some authors appear to use the term ‘metainterpreter’ carelessly, more or less as a synonym to ‘interpreter’, while others attribute other properties altogether to a metainterpreter, so some care is required.
queries. The classic three-line interpreter for a subset of Prolog in Prolog

\[
\begin{align*}
\text{solve}(\text{true}). \\
\text{solve}((P, Q)) & : \Rightarrow \text{solve}(P), \text{solve}(Q). \\
\text{solve}(H) & : \Rightarrow \text{clause}(H, B), \text{solve}(B).
\end{align*}
\]

(where \textit{Clause} is a predicate that generates representations of all applicable program clauses) uses both the unification and the backtracking of the underlying Prolog system. There are a number of theoretical and practical problems associated with this program and extensions to it, but it has is an enormous efficiency advantage over an interpreter that employs an explicit representation of substitutions etc.

The Gödel language (see Sect. 5.4) does not directly support a nonground representation (such interpreters can be written but the analogue of \textit{Clause} is missing). For a further discussion of the role of interpreters in logic programs, see below.

Kowalski has presented some justification [54] for the nonground representation, claiming that an interpreter using the nonground representation can be obtained by partial evaluation of an interpreter using a proper ground representation. In addition, De Schreye & Martens has presented a least Herbrand model semantics for what they call “untyped vanilla” metaprogramming, including a form of amalgamation [23]. These results are extended by Levi & Ramundo [57].

**Enhancing metainterpreters.** Sterling has proposed a methodology primarily intended for building knowledge systems [82], where a given ‘vanilla’ interpreter (essentially the three-line interpreter described previously) is enhanced with features such as generation of explanations (proofs), depth control etc. The idea is that each enhancement of the interpreter can be specified separately but that the enhancements subsequently can be combined with the vanilla interpreter, resulting in an interpreter that provides all given enhancements. The enhanced interpreter can then be applied to various object level knowledge bases.

5 MetaLogic Programming Languages

We present the most important proposed languages and other systems for metaprogramming in logic.

5.1 FOL

Weyhrauch’s FOL is an interactive first order logic proof system which incorporates reflection principles as inference rules [89]. Aiello, Nardi & Schaefer showed an example of how FOL could be used for expressing metareasoning (see Sect. 6.3) [2].
5.2 MetaProlog

The language MetaProlog is an extension of Prolog with metaprogramming capabilities [7]. The extensions are primarily aimed at enabling programs to manipulate multiple theories, allowing distinct theories to have multiple 'viewpoints' of a relation, and to make proofs available as terms in the language [3]. The MetaProlog compiler is able to compile theories also when they are only partially given. A preliminary declarative and operational semantics for MetaProlog was given by Subrahmanian [8].

5.3 Reflective Prolog

Reflective Prolog, which has been developed at Università degli Studi di Milano [20], allows the programmer to extend the object level provability relation by clauses to be used at the metalevel. However, a Reflective Prolog program is an amalgamated theory, containing object level axioms as well as metalevel axioms. For example, the clauses

solve(#P($X,$Y)) :- reflexive(#P), solve(#P($Y,$X)).
reflexive(<spouse>).

(where #P is a metalevel variable ranging over names of predicate symbols while $X$ and $Y$ are metalevel variables ranging over names of terms) say that a binary atom, whose predicate symbol denotes a reflexive relation, can be proved by instead proving an atom whose arguments have been exchanged with respect to the original atom; the predicate symbol spouse denotes such a relation.

One advantage with the form of metaprogramming supported by Reflective Prolog is that programs can be compiled to run with more or less the same speed as Prolog except when going between levels. It is certainly possible to write interpreters and program manipulation tools also in Reflective Prolog, by taking advantage only of the representation of expressions provided in the language.

5.4 Gödel

The Gödel language, developed at Bristol University by Hill, Lloyd et al., was originally intended as a logic programming language with strong metaprogramming features. During its development the ambitions have changed and the language is now proposed as an almost completely declarative successor of Prolog. In the current definition of Gödel [39] the metaprogramming capabilities are present through a collection of predefined modules. These modules define a ground representation of Gödel programs as abstract data types, a theory data type, and metalevel predicates that attempt to prove a formula. The language contains some support for a flat representation. There is also an intention to provide tools for building interpreters (for subsets of Gödel itself or other languages). The computational overhead
of such interpreters are hoped to be removed by partial evaluation of the interpreters, given the programs to be interpreted (see also Sect. 2.2, 3.1 and 6.1).

5.5 MOL

Eshghi, in his Ph.D. thesis, defined an amalgamated system called MOL. MOL includes a novel naming relation that gives structured names and includes constant symbols as names of variables. One particular feature of MOL is that it allows self-reference and Eshghi shows how this property can be used to simulate higher-order features and to obtain negation as failure. Eshghi also illustrates applications of MOL for metalevel control and metalevel reasoning.

5.6 Meta

Kowalski has proposed, largely indirectly through examples, a logic language with metaprogramming capabilities [55]. He proposes to name each theory by a constant, or possibly a more complex term, which does not encode the nonlogical axioms of the theory. The nonlogical axioms of a theory are instead given separately; each axiom is statically declared as belonging to some particular theory.

In fact, the declaration of nonlogical axioms is merely a special case of defining the theorems of a theory. The binary Demo predicate is defined by program clauses. The nonlogical axioms of a theory are declared by unit clauses, whose first argument is a term denoting a theory and whose second argument is the representation of a program clause. For example, the clause

\[
\text{Demo}(\text{John}, \text{Interesting}(\text{Mary}))
\]

declares that \text{Interesting}(\text{Mary}) is an 'explicit' theorem, i.e., an axiom, of the theory \text{John}. Inference rules of a theory are defined by non-unit clauses for Demo. For example, the clause

\[
\text{Demo}(\text{John}, p) \leftarrow \text{Demo}(\text{John}, p \leftarrow q) \land \text{Demo}(\text{John}, q)
\]

declares that \text{modus ponens} is an inference rule of the theory \text{john}. Note that no real distinction is made between a nonlogical axiom and an inference rule with no premises. This clause contains two variables \(p\) and \(q\) that are properly seen as metalevel variables or schema variables. Clauses for Demo containing variables in the theory argument define theorems of every theory whose name is an instance of the theory argument. For example, the clause

\[
\text{Demo}(t, \text{Interesting}(\text{Mary})) \leftarrow \text{Male}(t)
\]
declares that \text{Interesting}(\text{Mary}) is a theorem of any theory named by an expression \(t\) for which \text{Male} holds.
5.7 Combinatory Logic Programming

Combinatory languages are free of locally bound variables. Nilsson has observed that this makes combinatory languages particularly interesting for metaprogramming, because representation of a combinatory language will not lead to the implementation problems mentioned earlier [68]. Nilsson explores some of the potential of combinatory logic for metaprogramming by presenting a metainterpreter for a proposed combinator logic programming language.

5.8 Truth Predicates

Sato has investigated the use of a self-referential truth predicate, given semantics in three-valued logic, for metaprogramming [77]. Using a transformation method he proves that the interpreter is complete with respect to the three-valued semantics. Jiang goes even further [41], advocating a language with a nonground representation that does not distinguish between terms and sentences, based on an earlier and stricter language of Richards [74].

5.9 Alloy

Alloy is a metalogic programming language proposed by Barklund & Hamfelt [5, 6]. The language combines ideas from Reflective Prolog (automatic meta-to-object reflection as an inference rule) and Kowalski’s ‘Meta’ (the representation of theories and the application of meta-to-object reflection also to program clauses). The language has been used to express a multilayered representation of legal knowledge, suitable for incorporating multiple interpretations.

6 Applications of Metaprogramming in Logic

We can identify four main uses of metaprogramming in logic:

- Metaprogramming as a formalism for writing program manipulation tools, such as compilers, program transformers, debuggers and abstract interpreters.
- Metaprogramming for control of the procedural behaviour of logic programs.
- Metaprogramming for representing knowledge, reasoning about knowledge, reasoning about reasoning, etc.
- Metaprogramming for writing interpreters for languages, either to realize a new programming or knowledge representation language, or to add functionality to a language processor, for example, to make it generate proof terms for successful proofs.
The first three branches are largely distinct, so we will discuss them separately in what follows. The fourth branch is of a quite general nature and was discussed separately above (Sect. 4).

6.1 Program Manipulation

The most important requirement on a metaprogramming language for writing program manipulation tools is that it contains expressions that represent the linguistic elements of an object programming language. The metalanguage expressions need not have any similarity to the elements of the object language and could even be an abstract data type, whose internal representation is purposely hidden from the user, as in the Gödel language (see Sect. 5.4). This is usually sufficient for writing compilers and many program transformers. Tools for more sophisticated analysis of programs in an object language obviously require some ability to reason also about the provability relation of the object level.

Details on programming manipulation methods fall outside the scope of this overview so we will limit ourselves to a brief presentation of some important methods.

Program Transformation. Program transformation is concerned with automatic rewriting of programs to new programs that have some desired properties. Typically a program so obtained is required to be semantically equivalent to the original program. Common program transformation steps are the FOLD/UNFOLD transformations that rewrite a sentence to an atomic sentence or vice versa, using a set of applicable clauses. Such methods were first studied in the context of functional programs (recursive equations) by Burstall & Darlington [15] and applied to logic programs by, e.g., Hogger [46]. A transformation system that preserves the so-called least model semantics of logic programs was presented by Tamaki & Sato [85].

Partial Evaluation. Partial evaluation (in the context of logic programming also known as "partial deduction") is a form of program transformation but is worth discussing separately, because of its interesting applicability to interpreters and other metaprograms.

In recent years partial evaluation of logic programs has been the subject of much research. This is generally motivated by the expectation to speed up arbitrary programs, but in particular for removing the overhead involved when running programs using an interpreter. The idea is to consider an interpreter as being a metalevel program with two inputs: (the representations of) an object level program and an object level query. The significant observation is that the object level program is usually known beforehand. The idea is to specialize the interpreter for this particular object level program, in effect transforming the object level program to a metalevel program.
having the same semantics as the interpreter given the original object level program. In this way one can expect to eliminate most or all overhead caused by the extra level of interpretation.

There are many interesting applications of partial evaluation, for example, obtaining compiler-compilers etc., which can be obtained if the partial evaluation program can be applied to (a representation of) itself (so-called Futamura projections) [28, 42].

Some interesting examples of writing partial evaluators as logic programs, or applying partial evaluation to logic programs are given by Komorowski [47, 48], Kahn [45], Sterling [82], Takeuchi & Furukawa [84], Lloyd & Shepherdson [38], Jones, Gomard & Sestoft [44], and Sahlin [76].

**Program Analysis.** Program analysis attempts to extract properties of a program; knowledge of such properties can subsequently be used in a language processor in order to increase the time or space efficiency when running the program. Some examples of properties to be extracted for logic programs are mode information (whether an argument of a predicate is used for input or output), type information, aliasing information (hidden sharing between variables) and liveness information (the life time of the data structures representing terms). Present methods are largely based on abstract interpretation [21], a general method that has successfully been applied to logic programs, by e.g., Mellish [63], Jones & Sondervand [43], Debray & Warren [24], Marriott & Sondervand [61], Muthukumar & Hermenegildo [67], and Cousot & Cousot [22]. This is an area in particularly rapid development.

**Implicit Programs.** This is a recent development that seems interesting for software engineering in the future. Given a language with metalevel capabilities, (object level) programs could be specified implicitly through metalevel information, for example, in the form of metalevel programs. Such ideas have been put forward by, e.g., Kwok & Sergot [56], Costantini & Lanzarone [20], and Barklund & Hamfelt [6].

**Other Forms of Program Manipulation.** There are certainly many other kinds of metaprograms with a purpose to manipulate representations of programs, e.g., compilation, program synthesis, program verification, debugging, etc.

We shall not elaborate on these kinds of metaprograms here but will simply note that they will all benefit from a well-defined representation of the object programs in the metaprogramming language. Practical experience has shown that the nonground representation of object programs often employed in tools written in Prolog is dangerous, in the sense that there is
a high risk that the programmer introduces subtle errors by unintentional instantiation of metavariables that actually represent object variables.

6.2 Metalevel Control

In the area of expert systems metalevel information has been used chiefly for controlling an inference process, for example, to heuristically choose the most promising line of reasoning. When the inference process is carried out within the framework of computational logic, this control should be seen as a form of metalogic programming. Two examples of such systems are Hayes’ GOLUX [35] and Genesereth’s MRS [31].

Sterling defined two metalevels beyond the domain (object) level [81] and claimed that knowledge on all three levels was necessary and probably sufficient for building knowledge systems, partly in order to control the lower level deductions.

The same idea has been applied for controlling the execution of logic programs in general. In Prolog a limited control of this kind has often been mixed in a single level. Primitives such as \texttt{var} can be used to partially inspect the state of the computation and fail undesirable computation paths. Gallaire & Lasserre proposed more metalevel primitives for fine-grained control of logic programs [29]. This metalevel knowledge was expressed as program clauses for certain reserved predicates. Another approach, based on the use of interpreters, was put forward by Pereira [69].

6.3 Knowledge Representation and Reasoning

Perhaps the most intriguing application of metalogic programming is in the area of knowledge representation. Techniques for representing knowledge in logic have been studied for a long time [27, 36, 62] and knowledge expressed in efficiently executable subsets of logic, e.g., in Horn clauses, constitutes a logic program having both a declarative and a procedural reading [51].

However, in order to handle declaratively and elegantly more than ‘static’ explicit knowledge, e.g., knowledge that evolves over time or is given only implicitly, perhaps through vague and incomplete descriptions, we may benefit from allowing also knowledge belonging to various metalevels to be part of logic programs.

A metalogical view of knowledge structures. Many proposed data structures for knowledge representation, most importantly frames but also, e.g., semantic nets and scripts, can be given a useful new logical reading when expressed in a metalogic programming language. The first such observation is due to Hayes [37], later Bowen [8] described several such correspondences in some detail. Further work along this direction is reported by Brogi & Turini [11].
**Three Wise Men.** The ‘three wise men’ problem has become a standard test problem for metareasoning and metaprogramming formalisms and systems, because it involves agents reasoning about the beliefs and reasoning of other agents.\(^9\) Briefly, a common version of the problem can be stated as follows. There is a king who wishes to test the cleverness of the kingdom’s three wisest men. They stand in a circle, so they can all see and hear each other as well as the king. The king places a hat on each wise man, telling them that every hat is white or black and that at least one hat is white. Actually, all hats are white. No wise man can see the colour of his own hat. The king then proceeds to ask each wise man, in turn, if he knows the colour of his hat. The first wise man answers that he does not know the colour of his hat. Then the second wise man answers the same. Finally, the third wise man answers that he knows his hat is white. The challenge is to represent the beliefs of (at least) the third wise man, based on what he has seen and heard from the king and the other wise men, and to carry out the reasoning leading up to the conclusion that his own hat must be white. Apparently, a minimal requirement of the belief representation formalism is that it can represent the fact that some wise man was unable to determine the colour of his hat.

One reasonably straightforward line of reasoning that the third wise man could perform is the following, where the information that at least one hat is white is incorporated by assuming that any wise man can reason: ‘if I only see black hats, then my own hat must be white’. The first man’s answer implies that he does not see only black hats. This information is received by the second and third wise men. Therefore the answer of the second wise man must be considered in a context where he knows that either his hat or the third wise man’s hat is white. If he had seen the third wise man wearing a black hat then he would have answered that he knew the colour of his hat. He did not, and therefore the third wise man can conclude that his hat must be white.

This argument can be constructed using metaprogramming as follows. Let the language of the third wise man be \(L_3\), which is assumed to represent the language \(L_2\) of the second wise man. This language is, in turn, assumed to represent the language \(L_1\) of the first wise man. The theory of the third wise man would have to contain theorems of the form \(\text{Demo}(w_2, \Phi)\), where \(w_2\) is an expression representing the third wise man’s view of one of the second wise man’s beliefs (\(\Phi\)). Some of these theorems would actually be of the form \(\text{Demo}(w_2, \text{Demo}(w_1, \Psi'))\), where \(w_1\) is an expression representing the second wise man’s view of one of the first wise man’s beliefs (\(\Psi\)), as viewed by the third wise man. In order to represent the beliefs exactly it

---

\(^9\)Below we will also say that agents ‘know’ things; for these purposes we can think of such knowledge as justified and true beliefs.
would be necessary to distinguish between the wise men’s beliefs at different points in time, because their available information increases as they hear each other’s answers to the king’s questions.

It is very instructive to compare various proposed solutions to this problem, although lack space prevents us from doing so here. They all contain some form of metareasoning in logic, but use different formal systems, including various forms of metaprogramming in logic [2, 19, 39, 46, 83] and modal logic [30, 32].

**Press and Impress.** Another well-known example of metalevel reasoning based on logic programming techniques is the Press system, developed at University of Edinburgh [14] (a retrospective view has been given by Welham [88]). The system solves equations by reasoning at the metalevel. The Impress system, in addition, employs the metametalevel in order to create proof plans for the metalevel [13].

**Multiple Interpretations.** By exploiting metaprogramming it is possible to construct knowledge systems for domains where the direct representations of knowledge must be the object of extensive interpretation, when applying it to actual problem situations. For example, knowledge represented at the object level can be subject to arbitrarily complex metalevel transformation and filtering before actually being used. This can be applied recursively, so metalevel knowledge (that may be part of a metalevel program manipulating object level knowledge) is, in turn, subject to interpretation by metametalevel knowledge. Such flexible interpretation is necessary for proper treatment of legal knowledge. This was first observed, and expressed in an executable formalism, by Hamfelt & Barklund [5, 6, 34]. Such techniques should be applicable also in other domains where reasoning cannot be performed by naive application of knowledge to situations and problems.

7 **Related Topics**

Metalogic programming has close relationships with several other logics that have been proposed for computation and/or knowledge representation. Two of the most interesting such logics are second-order logic and epistemic or doxastic modal logics and we shall briefly review these relationships.

**Second Order Logic.** The crucial distinction between a metalogic and a second-order logic is that the former contains variables ranging over representations of symbols, expressions and proofs, while the latter quantifies over actual relations, e.g., subsets of cartesian products of the universe.

We change no important properties of a language by letting some of its ground expressions represent expressions in another language, or even
expressions in the language itself. Second-order languages, on the other hand, because of their higher expressibility,\(^\text{10}\) lack some fundamental properties that first-order languages have and there are no general efficient proof procedures for such languages.

In application areas of metalogic programming where the language representation capabilities are crucial, e.g., when writing a program analyzer, second order logic is not an alternative. However, in other areas, most importantly knowledge representation, we see a use for both metalogic and second-order logic. For example, we require metalogical sentences when we want to perform reasoning that depends also on the form of the sentences in the knowledge base or the query, or when representations of proofs need to be constructed and analyzed. On the other hand, if we want a knowledge base to contain a certain function or relation, expressed in terms of other functions and relations, then it might be significantly more convenient to express the new relation through (set) operations on the existing relations.

Higher-order logic programming in the form of a language called Prolog has been investigated by Miller & Nadathur [64, 65].

Achieving a more precise understanding of the relative merits of metalogic and second order logic, particularly in applications, is an interesting open research problem.

Epistemic or Doxastic Modal Logics. A modal logic for beliefs [49] is a logic where belief is expressed through a truth-valued operator that takes a truth-valued expression as argument, e.g., \([\text{Bill} \text{ Funny}(\text{Three Stooges})]\) rather than through a predicate symbol that takes the representation of a truth-valued expression as argument, e.g., \(\text{Demo}_{\text{Bill}}(\text{Funny}(\text{Three Stooges}))\).

In many ways, a modal logic is easier to use for such purposes, since it is not necessary to work with possibly complex encodings of expressions, but simply with the expressions themselves. On the other hand, this simplicity is paid for in that modal logics lack many useful properties of classical logic (and metalogic). For example, when using so-called quantifying-in, i.e., an expression where a variable quantified outside a modal expression is referred to from inside the modal expression, e.g., the expression \(\exists x([\text{Bill} \text{ Funny}(x)])\), applying skolemization yields the same result, e.g., \([\text{Bill} \text{ Funny}(C)]\), as for the quite different sentence \([\text{Bill} \exists x(\text{Funny}(x))\). Such problems lead to strong criticism from Quine [73].

Konolige has proposed a modal logic \(\mathbf{B}\), with language \(L^\mathbf{B}\), for beliefs and a computational procedure for answering queries in this language [49].

\(^{10}\)Second-order languages have variables that range over a strictly richer universe than those of first-order languages, so there exist propositions that can be expressed in second-order languages but not in first-order languages.
8 Future Developments

Metaprogramming in logic has a potential that has barely begun to be exploited. It has advantages over metaprogramming using conventional programming languages in that the theoretical issues are much clearer, having been studied even before computing as we know it was invented. It is also richer than metaprogramming in conventional languages, because concepts such as provability are naturally present, and deeper because problems with negation are addressed.

Metaprogramming as such is necessary when constructing program manipulation tools and when realizing languages through interpreters. As mentioned above, the majority of work so far in metalogic programming has been using an improper nonground representation. In the future the ground representations have the greater potential, because such representations behave as ordinary data structures and carry a maximum of explicit information. Efficient implementation of such representations is presently being given significant attention and it seems certain that, as in most other areas of computing, the advantages of using improper methods vanish over time.

In the area of reasoning, metareasoning and knowledge representation it is expected that an increased understanding of reasoning about provability in formal languages will shed light on many present problems that cannot be expressed accurately in single level formalisms or that cannot be computed efficiently. Many objections against using logic and other declarative formalisms for knowledge representation are valid for single-level systems but do not apply to systems with metalogical capabilities.

Finally, although formal techniques for metaprogramming have probably been developed furthest in computational logic, it would make sense to apply similar techniques also in more conventional computation. Computational introspection and proper representations of programs should be valuable for developing correct and efficient tools for software engineering.

Further Reading

In addition to the literature references in the preceding text, the following bibliography includes four collections of articles [1, 12, 60, 72] and two books [32, 49] that can be consulted by the interested reader to explore this topic further.

Acknowledgements

I would like to thank professors N. D. Jones, J. W. Lloyd, and J. F. Nilsson for valuable comments on this essay, and my colleagues and students for stimulating discussions and comments.

The compilation of this survey was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under contract No. 92-10452 (ESPRIT BRP 6810: Computational Logic 2).
Bibliography


