

University of Leeds
SCHOOL OF COMPUTER STUDIES
RESEARCH REPORT SERIES
Report 94.22

Meta-Programming in
Logic Programming¹

by

P M Hill & J G Gallagher²
Division of Artificial Intelligence

August 1994

¹To be published in Volume V of the Handbook of Logic in Artificial Intelligence and Logic Programming, Oxford University Press.

²Department of Computer Science, University of Bristol.

Meta-Programming in Logic Programming

P. M. Hill and J. Gallagher

Contents

1	Introduction	2
	1.1 Theoretical Foundations	3
	1.2 Applications	5
	1.3 Efficiency Improvements	6
	1.4 Preliminaries	6
2	The Non-Ground Representation	9
	2.1 The Representation	11
	2.2 Reflective Predicates	14
	2.3 Meta-Programming in Prolog	19
3	The Ground Representation	20
	3.1 The Representation	22
	3.2 Reflective Predicates	28
	3.3 The Language Gödel and Meta-Programming	32
4	Self-Applicability	38
	4.1 Separated Meta-Programming	40
	4.2 Amalgamated Meta-Programming	41
	4.3 Ambivalent Logic	47
5	Dynamic Meta-Programming	48
	5.1 Constructing Programs	48
	5.2 Updating Programs	50
	5.3 The Three Wise Men Problem	53
	5.4 Transforming and Specializing Programs	57
6	Specialization of Meta-Programs	60
	6.1 Logic Program Specialization	61
	6.2 Specialization and Compilation	66
	6.3 Self-Applicable Program Specializers	67
	6.4 Applications of Meta-Program Specialization	69

1 Introduction

A meta-program, regardless of the nature of the programming language, is a program whose data denotes another (object) program. The importance of meta-programming can be gauged from its large number of applications. These include compilers, interpreters, program analysers, and program transformers. Furthermore, a logic program when used in artificial intelligence often formalises some knowledge; in this case a meta-program is viewed as a meta-reasoner for reasoning about this knowledge.

We have identified three major topics for consideration in this chapter on meta-programming. These are the theoretical foundations of meta-programming, the suitability of the alternative meta-programming techniques for different applications, and methods for improving the efficiency of meta-programs. As with logic programs generally, meta-programs have declarative and procedural semantics. The theoretical study of meta-programming shows that both aspects of the semantics depend crucially on the manner in which object programs are represented as data in a meta-program. The second theme of the paper is the problem of designing and choosing appropriate ways of specifying important meta-programming problems, including dynamic meta-programming and problems involving self-application. The third theme concerns efficient implementation of meta-programs. Meta-programming systems require representations with facilities that minimize the overhead of interpreting the object program. In addition, efficiency can be gained by transforming the meta-program, specializing it for the particular object program it is reasoning about. This chapter, which concentrates on these aspects of meta-programming, is not intended to be a survey of the field. A more complete survey of meta-programming for logic programming can be found in (Barklund 1994).

Many issues in meta-programming have their roots in problems in logic which have been studied for several decades. This chapter emphasizes meta-programming solutions, and is not intended to give a full treatment of the underlying logical problems, though we try to indicate some connections to wider topics in meta-logic.

To avoid confusion between a programming language such as Prolog and the language of an actual program, the programming language will be referred to as a programming *system* and the word *language* will be used to refer to the set of expressions defined by a specific alphabet together with rules of construction. Note that the programming system for the object program does not necessarily have to be the same as the programming system for the meta-program although this is often the case and most theoretical work on meta-programming in logic programming makes this assumption.

1.1 Theoretical Foundations

The key to the semantics of a meta-program is the way the object program expressions are represented in the meta-program. However in logic programming there is no clear distinction between the data and the program, since the data is sometimes encoded as program clauses, and the question arises as to what kind of image of the object program can be included in the meta-program. Normally, a representation is given for each symbol of the object language. This is called a *naming relation*. Then rules of construction can be used to define the representation of the constructed terms and formulas. Each expression in the language of the object program should have at least one representation as an expression in the language of the meta-program.

It is straightforward to define a naming relation for the constants, functions, propositions, predicates and connectives in a language. For example, we can syntactically represent each object symbol s by its primed form s' or even by the same token s . However, although a meta-program symbol may be syntactically similar to the object symbol it represents, it may be in a different syntactic category. Thus a predicate is often represented as a function and a proposition as a constant. A connective in the object language may be represented as a connective, predicate, or function in the meta-program.

The main problem arises in the representation of the variables of the object language in the language of the meta-program. There are two options; either represent the object variables as ground terms or represent them as variables (or, more generally, non-ground terms). The first is called the *ground representation* and the second the *non-ground representation*. In logic, variables are normally represented as ground terms. Such a representation has been shown to have considerable potential for reasoning about an object theory. For example, the arithmetization of first order logic, illustrated by the Gödel numbering, has been used in first order logic to prove the well-known completeness and incompleteness theorems. As logic programming has the full power of a Turing machine, it is clearly possible to write declarative meta-programs that use the ground representation. However, ease of programming and the efficiency of the implementation are key factors in the choice of technique used to solve a problem. So the support provided by a programming language for meta-programming often determines which representation should be used. Most logic programming systems have been based on Prolog and this has only provided explicit support for the representation of object variables as variables in the language of the meta-program. It was clear that there were several semantic problems with this approach, and as a consequence, for many years, the majority of meta-programs in logic programming had no clear declarative semantics and their only defined semantics was procedural. The situation

is now better understood and the problem has been addressed in two ways.

One solution is to clarify the semantics of the non-ground representation. This has been done by a number of researchers and the semantics is now better understood. The other solution is for the programmer to write meta-programs that use the ground representation. To save the user the work of constructing a ground representation, systems such as Gödel (Hill & Lloyd 1994) and Reflective Prolog (Costantini & Lanzarone 1989) that provide a built-in ground representation have been developed. There are advantages and disadvantages with each of these solutions. It is much more difficult to provide an efficient implementation when using the ground representation compared to the non-ground representation. However, the ground representation is far more expressive than the non-ground representation and can be used for many more meta-programming tasks. In Sections 2 and 3, we discuss the non-ground and ground representations, respectively, in more detail.

One other issue concerning the syntactic representation is how the theory of the object program is represented in the meta-program. The meta-program can either include the representation of the object program it is reasoning about as program statements, or represent the object program as a term in a goal that is executed in the meta-program. In the first case the components of the object program are fixed and the meta-program is specialized for just those object programs that can be constructed from these components. In the second case, the meta-program can reason about arbitrary object programs. These can be either fixed or constructed dynamically during the derivation procedure.

The syntactic representation of an object program as outlined above ignores the semantics of object expressions. As an example, consider an object program that implements the usual rules of arithmetic. In such a program, $1 + 1$ and 2 should denote the same value. Assuming a simple syntactic representation, a meta-program would normally represent these expressions as distinct terms so that, unless the rules of arithmetic were attached to the representation, these would not denote the same value in the meta-program. In meta-reasoning, a relation defining such an attachment is called a *reflection principle*.

In logic programming, a meta-program will normally require a variety of *reflective predicates* realising different reflective principles. For example, there may be a reflective predicate defining SLD-resolution for (the representation of) the object program. Other reflective predicates may define (using a representation of the object program) unification or a single derivation step. In fact, it is these more basic steps for which support is often required.

To make meta-programming a practical tool, many useful reflective predicates (based on a pre-defined representation) are often built into the programming system. However, a programming system cannot provide all

possible reflective predicates even for a fixed representation. Thus a system that supports a representation with built-in reflective predicates must also provide a means by which a user can define additional reflective predicates appropriate for the particular application. The actual definition of the reflective predicates depends not only on the reflective principles it is intended to model but also on the representation. Later in this chapter, we discuss three reflective predicates in detail. One is *Solve/2* which is defined for the non-ground representation and the others are *IDemo/3* and *JDemo/3* which are defined for the ground representation. Both Prolog and Gödel provide system predicates that are reflective. For example, in Prolog, the predicate `call/1` is true if its only argument represents the body of a goal and that goal is true in the object program. The Gödel system provides the predicate `Succeed` which has three arguments. `Succeed/3` is true if the third argument represents the answer that would be computed by the Gödel program represented in the first argument using the goal represented in the second argument.

1.2 Applications

We identify two important application requirements in this chapter. One is for meta-programs that can be applied to (representations of) themselves and the other is for meta-programs that need to reason about object programs that can change. We call the first, *self-applicable*, and the second, *dynamic* meta-programming.

There are many programming tools for which self-application is important. In particular; interpreters, compilers, program analysers, program transformers, program debuggers, and program specializers can be usefully applied to themselves. Self-applicable meta-programming is discussed in Section 4 and the use of self-applicable program specializers is discussed in Section 6.

For static meta-programming, where the object program is fixed, the meta-program can include the representation of the object program it is reasoning about in its program statements. However, frequently the purpose of the meta-program is to create an object program in a controlled way. For example, the object program may be a database that must change with time as new information arrives; the meta-program may be intended to perform hypothetical reasoning on the object program; or the object program may consist of a number of components and the meta-program is intended to reason with a combination of these components. This form of meta-programming in which the object program is changed or constructed by the meta-program we call *dynamic*. Section 5 explains the different forms of dynamic meta-programming in more detail.

Of course there are many applications which are both self-applicable and dynamic and for these we need a combination of the ideas discussed in these sections. One of these is a program that transforms other programs. As the

usual motivation for transforming a program is to make its implementation more efficient, it clearly desirable for the program transformer to be self-applicable. Moreover, the program transformer has to construct a new object program dynamically, possibly interpreting at different stages of the transformation (possibly temporary) versions of the object programs. We describe such an application in Section 6.

1.3 Efficiency Improvements

Despite the fact that meta-programming is often intended to implement a more efficient computation of an object program, there can be a significant loss of efficiency even when the meta-program just simulates the reasoning of the object program. This is partly due to the extra syntax that is required in the representation and partly due to the fact that the compiler for the object program performs a number of optimizations which are not included in the meta-program's simulation. One way of addressing this problem is through program specialization. This approach is explained in Section 6. The main aim of specialization is to reduce the overhead associated with manipulating object language expressions. When dealing with a fixed object theory the overhead can largely be “compiled away” by pre-computing, or “partially evaluating” parts of the meta-program's computation.

A second reason for considering specialization is that it can establish a practical link between the ground and the non-ground representations. In certain circumstances a meta-program that uses the non-ground representation can be obtained by partially evaluating one that uses a ground representation.

1.4 Preliminaries

The two principal representations, non-ground and ground, discussed in this chapter are supported by the programming systems, Prolog and Gödel, respectively. Since Prolog and Gödel have different syntax, we have, for uniformity, adopted a syntax similar to Gödel. Thus, for example, variables begin with a lower-case letter, non-variable symbols are either non-alphabetic or begin with an upper-case letter. The logical connectives are standard. Thus \wedge , \neg , \leftarrow , and \exists denote conjunction, negation, left implication and existential quantification, respectively. The exception to the use of this syntax is where we quote from particular programming systems. In these cases, we adopt the appropriate notation.

Figures 1 and 2 contain two simple examples of logic programs (in this syntax) that will be used as object programs to illustrate the meta-programming concepts in later sections. There are two syntactic forms for the definition of *Member* in Figure 2. One uses the standard list notation [...|...] and the other uses the constant *Nil* and function *Cons/2* to construct a list. For illustrating the representations it is usually more informative

```

Cat(Tom)
Mouse(Jerry)
Chase(x,y) ← Cat(x) ∧ Mouse(y)

```

Fig. 1. The Chase Program

```

Member(x, [x|y])
Member(x, [z|y]) ← Member(x, y)

Member(x, Cons(x, y))
Member(x, Cons(z, y)) ← Member(x, y)

```

Fig. 2. The Member Program

to use the latter form although the former is simpler. The language of the program in Figure 1 is assumed to be defined using just the symbols *Cat/1*, *Mouse/1*, *Chase/2*, *Tom*, and *Jerry* occurring in the program. The language of the program in Figure 2 is assumed to include the non-logical symbols in the program in Figure 1, the predicate *Member/2*, the function *Cons/2*, the constant *Nil*, together with the natural numbers.

We summarise here the main logic programming concepts required for this chapter. Our terminology is based on that of (Lloyd 1987) and the reader is referred to this book for more details. A *logic program* contains a set of program statements. A *program statement* which is a formula in first order logic is written as either

H

or

$H \leftarrow B$

where H is an atom and B is an arbitrary formula called the *body* of the statement. If B is a conjunction of literals (respectively, atoms), then the program statement is called a *normal* (respectively, *definite*) clause. A program is *normal* (respectively, *definite*) if all its statements are normal (respectively, definite). A *goal* for a program is written as

$\leftarrow B$

denoting the formula $\neg B$, where B is an arbitrary formula called the *body* of the goal. If B is a conjunction of literals (respectively, atoms), then the goal is *normal* (respectively, *definite*). As in Prolog and Gödel, an ‘_’ is used to denote a unique variable existentially quantified at the front of the atom in which it occurs. It is assumed that all other free variables are universally quantified at the front of the statement or goal.

The usual procedural meaning given to a definite logic program is SLD-resolution. However, this is inadequate to deal with more general types of program statements. In logic programming, a (ground) negative literal $\neg A$ in the body of a normal clause or goal is usually implemented by ‘negation

as failure’:

the goal $\neg A$ succeeds if A fails

the goal $\neg A$ fails if A succeeds.

Using the program as the theory, negation, as defined in classical logic, cannot provide a semantics for such a procedure. However, it has been shown that negation as failure is a satisfactory implementation of logical negation provided it is assumed that the theory of a program is not just the set of statements in the program, but is its *completion* (Clark 1978). A clear and detailed account of negation as failure and a definition of the completion of a normal program is given in (Shepherdson 1994). This definition can easily be extended to programs with arbitrary program statements. Moreover, it is shown in (Lloyd 1987) that any program can be transformed to an equivalent normal program. Thus, in this chapter, we only consider object programs that are normal and, unless otherwise stated, that the semantics of a logic program is defined by its completion. The completion of a program P is denoted by $\text{comp}(P)$.

In meta-programming, we are concerned, not only with the actual formulas in an object theory but also the language in which these formulas are written. This language which we call the *object language* may either be inferred from the symbols appearing in the object theory, or be explicitly declared. When the object theory is a Prolog program, then the language is inferred. However, when the object program is a Gödel program the language is declared. These declarations also declare the types of the symbols so that the semantics of the Gödel system is based on many-sorted logic.

For the following discussion we adopt some notation:

1. The symbols \mathcal{L} and \mathcal{M} denote languages. Usually \mathcal{L} denotes an object language while \mathcal{M} is the language of some meta-program.
2. The notation $E_{\mathcal{L}}$ means that E is an expression in a language \mathcal{L} . We omit the subscript when the language is either obvious or irrelevant.
3. The representation of an expression E in another language \mathcal{M} is written $[E]_{\mathcal{M}}$. The actual representation intended by this notation will depend on the context in which it is used. We omit the subscript when the language is either obvious or irrelevant.
4. If \mathcal{L} is a language, then the set of representations of expressions of \mathcal{L} in language \mathcal{M} is written $[\mathcal{L}]_{\mathcal{M}}$.

We will discuss a number of reflective predicates, but the key examples will realise adaptations of one or both of the following reflective principles. For all finite sets of sentences A and single sentences B of an object language \mathcal{L} ,

$$A \vdash_{\mathcal{L}} B \text{ iff } Pr \vdash_{\mathcal{M}} Demo([A], [B])$$

$$A \models_{\mathcal{L}} B \text{ iff } Pr \models_{\mathcal{M}} Demo([A], [B])$$

where Pr denotes the theory of the meta-program with language \mathcal{M} and $Demo/2$ denotes the reflective predicate. Such a program is often called a *meta-interpreter*.

A meta-program reasons about another (object) program. As this object program can itself be another meta-program, a *tower* of meta-programs can be constructed. Within such a tower, a meta-program can be assigned a *level*. The base level of the tower will be an object program, say P_0 , which is not a meta-program. At the next level, a program P_1 will be a meta-program for reasoning about P_0 . Similarly, if, for each $i \in \{1, \dots, n\}$, P_{i-1} is an object program for P_i , then P_i is one level above the level of P_{i-1} . Normally, we are only concerned in any two consecutive levels and refer to the relative positions of an object program and its meta-program within such a tower as the *object level* and *meta-level*, respectively.

2 The Non-Ground Representation

The non-ground representation requires the variables in the object program to be represented by non-ground terms in the meta-program. This section assumes, as in Prolog, that object variables are represented as variables in the meta-program. Before discussing the representation in more detail, we describe the historical background to this representation.

In the early work on logic programming it appears that there was an understanding that the Prolog system being developed should be self-applicable. That is, it should facilitate the interpretation of Prolog in Prolog. The first implementation of a programming system based on the ideas of logic programming was Marseille Prolog (Colmerauer, Kanoui, Pasero & Roussel 1973). This system was designed for natural language processing but it still provided a limited number of meta-programming features. However, the meta-programming predicates such as **clause**, **assert**, and **retract** that we are familiar with in Prolog (and now being formally defined by the committee for the ISO standard for Prolog) were introduced by DHD Warren and were first included as part of DEC-10 Prolog (Pereira, Pereira & Warren 1978). Moreover, the ability to use the meta-programming facilities of the Prolog system to interpret Prolog was first demonstrated in (Pereira et al. 1978) with the following program.

```
execute(true).
execute((A,B)) :- execute(A), execute(B).
execute(A) :- clause(A,B), execute(B).
execute(A) :- A.
```

The program shows the ease by which Prolog programs can interpret themselves. It was stated that the last clause enabled the interpreter “to cope with calls to ordinary Prolog predicates”. It is not clear here what is meant by “ordinary” predicates, but we assume that the main purpose for this

clause was to enable the interpreter to cope with system predicates. The use of the variable **A** instead of a non-variable goal can be avoided by means of the **call** predicate which was also provided by DEC-10 Prolog. Thus the last clause in this interpreter could have been written as

```
execute(A) :- call(A).
```

Both the use of a variable for a goal and the provision of the **call** predicate corresponded to and was (probably) inspired by the *eval* function of Lisp.

Clark and McCabe (1979) illustrated how Prolog could be used for implementing expert systems. They made extensive use of the meta-programming facility that allows the use of a variable as a formula. Here the variable also occurred in the head of the clause or in an atom in the body of the clause to the left of the variable formula. The programmer had to ensure that the variable was adequately instantiated before the variable formula was called. For example in:

```
r(C) :- system(C), C.
```

either the call to **r** should have an atomic formula as its argument, or the predicate **system/1** would have to be defined so that it bound **C** to an atomic formula. The left to right computation rule of Prolog ensured that **system(C)** was called before **C**.

This meta-variable feature can be explained as a schema for a set of clauses, one for each relation used in the program. This explanation is credited to Roussel. Using just this meta-programming feature, Clark and McCabe showed how a proof tree could be constructed in solving a query to a specially adapted form of the expert system. A disadvantage of this approach was that the object program had to be explicitly modified by the programmer to support the production of the proof as a term.

Shapiro (1982) developed a Prolog interpreter for declarative debugging that followed the pattern of the **execute** program above. This was a program which was intended for debugging logic programs where the expected (correct) outcome of queries to a program is supplied by the programmer. The programmer can then ignore the actual trace of the execution. The meta-program was based on an interpreter similar to the above program from (Pereira et al. 1978) but with the addition of an atom **system(A)** as a conjunct to the body of the last statement. **system/1** is intended to identify those predicates for which **clause/2** is undefined or explicit interpretation is not required.

A major problem with the use of meta-programs for interpreting other Prolog programs is the loss of efficiency caused by interpreting the object programs indirectly through a meta-program. Gallagher in (1986) and Takeuchi and Furukawa in (1986) showed that most of the overhead due to meta-programming in Prolog could be removed by partially evaluating the meta-program with respect to a specific object program. This important development and other research concerning program transformations en-

couraged further development of many kinds of meta-programming tools. For example, Sterling and Beer (1986) and (1989) developed tools for transforming a knowledge base together with a collection of meta-programs for interpreting an arbitrary knowledge base into a program with the functionality of all the interpreters specialized for the given knowledge base.

It is clear that many of the meta-programming facilities of Prolog such as the predicates `var`, `nonvar`, `assert`, and `retract` do not have a declarative semantics. However, the predicates such as `clause` and `functor` are less problematical and can be interpreted declaratively. In spite of this, the semantics of these predicates and a logical analysis of simple meta-programs that used them was not published until the first workshop on meta-programming in logic in 1988 (Abramson & Rogers 1989). Substantial work has been done since on the semantics of the Prolog style of meta-programming and some of this will be discussed later in this section. Since a proper theoretical account of any meta-program depends on the details of how the object program is represented, in the next subsection we describe the simple non-ground representation upon which the Prolog meta-programming provision is based.

2.1 The Representation

It can be seen from the above historical notes that the main motivation for a non-ground representation is its use in Prolog. Hence, the primary interest is in the case where variables are represented as variables and non-variable symbols are represented by constants and functions in the language of the meta-program.

Therefore, in this section, a non-ground representation is presented where variables are represented as variables and the naming relation for the non-logical symbols is summarised as follows.

Object Symbols	Meta Symbols
Constant	Constant
Function of arity n	Function of arity n
Proposition	Constant
Predicate of arity n	Function of arity n

Distinct symbols (including the variables) in the object language must be named by distinct symbols in the meta-language. For example, *Tom* and *Jerry* in the Chase program in Figure 1 could be named by constants, say *Tom'* and *Jerry'*. The predicates *Cat/1*, *Mouse/1*, and *Chase/2* can be similarly represented by functions, say *Cat'/1*, *Mouse'/1*, and *Chase'/2*.

For a given naming relation, the representations $[t]$ and $[A]$ of a term t and atom A , are defined as follows.

- If t is a variable x , then $[t]$ is the variable x .
- If t is a constant C , then $[t]$ is the constant C' , where C' is the name

of C .

- If t is the term $F(t_1, \dots, t_n)$, then $[t]$ is $F'([t_1], \dots, [t_n])$, where F' is the name of F .
- If A is the atom $P(t_1, \dots, t_n)$, then $[A]$ is $P'([t_1], \dots, [t_n])$, where P' is the name of P .

For example, the atom $Cat(Tom)$ is represented (using the above naming relation) by $Cat'(Tom')$ and $Mouse(x)$ by $Mouse'(x)$. To represent non-atomic formulas, a representation of the logical connectives is required. The representation is as follows.

Object Connectives	Meta Symbols
Binary connective	Function of arity 2
Unary connective	Function of arity 1

We assume the following representation for the connectives used in the examples here.

Object Connective	Representation
\neg	Prefix function <i>Not</i> /1
\wedge	Infix function <i>And</i> /2
\leftarrow	Infix function <i>If</i> /2

Given a representation of the atomic formulas and the above representation of the connectives, the term $[Q]$ representing a formula Q is defined as follows.

- If Q is of the form $\neg R$, then $[Q]$ is *Not* $[R]$.
- If Q is of the form $R \wedge S$, then $[Q]$ is $[R]$ *And* $[S]$.
- If Q is of the form $R \leftarrow S$, then $[Q]$ is $[R]$ *If* $[S]$.

Continuing the above example using this naming relation, the formula $Cat(Tom) \wedge Mouse(Jerry)$

is represented by the term

$Cat'(Tom') \text{ And } Mouse'(Jerry')$.

In this example, the name of an object symbol is distinct from the name of the symbol that represents it. However, there is no requirement that this should be the case. Thus, the names of the object symbol and the symbol that represents it can be the same. For example, with this representation, the atomic formula $Cat(Tom)$ is represented by the term $Cat(Tom)$. This is the trivial naming relation, used in Prolog. It does not in itself cause any amalgamation of the object language and meta-language; it is just a syntactic convenience for the programmer. We adopt this trivial naming relation together with the above representation of the connectives for the rest of this section.

A logic program is a set of normal clauses. It is clearly necessary that if the meta-program is to reason about the object program, there must be a way of identifying the clauses of a program. In Prolog, each clause in the program is represented as a fact (that is, a clause with the empty body). This has the advantage that the variables in the fact are automatically standardized apart each time the fact is used. We adopt this representation here. Thus it is assumed that there is a distinguished constant *True* and distinguished predicate *Clause/2* in the meta-program defined so that each clause in the object program of the form

$$h \leftarrow b.$$

is represented in the meta-program as a fact

$$\text{Clause}([h], [b]).$$

and each fact

$$h.$$

is represented in the meta-program as a fact

$$\text{Clause}([h], \text{True}).$$

Thus, in Figure 1,

$$\text{Chase}(x, y) \leftarrow \text{Cat}(x) \wedge \text{Mouse}(y).$$

is represented by the fact

$$\text{Clause}(\text{Chase}(x, y), \text{Cat}(x) \text{ And } \text{Mouse}(y)).$$

The program in Figure 2 would be represented by the two facts.

$$\text{Clause}(\text{Member}(x, \text{Cons}(x, -)), \text{True}).$$

$$\text{Clause}(\text{Member}(x, \text{Cons}(-, y)), \text{Member}(x, y)).$$

An issue that has had much attention is whether the facts defining *Clause* accurately represent the object program. The problem is a consequence of the fact that, in Prolog, the language is not explicitly defined but assumed to be determined by the symbols used in the program and goal. Thus, the variables in the object program range over the terms in the language of the object program while the variables in the definition of *Clause* range, not only over the terms representing terms in the object program, but also over the terms representing the formulas of the object program. Thus, in Figure 1, the terms in the object language are just the two constants *Tom* and *Jerry*, while in a meta-program representing this program, the terms not only include *Tom* and *Jerry* but also *Cat(Tom)*, *Mouse(Jerry)*, *Cat(Cat(Tom))* and so on. Thus in the clause

$$\text{Chase}(x, y) \leftarrow \text{Cat}(x) \wedge \text{Mouse}(y).$$

in the object program, *x* and *y* are assumed to be universally quantified in a domain just containing *Tom* and *Jerry*, while in the fact

$$\text{Clause}(\text{Chase}(x, y), \text{Cat}(x) \text{ And } \text{Mouse}(y)).$$

in the meta-program, x and y are assumed to be universally quantified in a domain that contains a representation of all terms and formulas of the object program.

There is a simple solution, that is, assume that the intended interpretation of the meta-program is typed. The types distinguish between the terms that represent terms in the object program and terms that represent the formulas. This approach has been developed by Hill and Lloyd (1989). An alternative solution is to assume an untyped interpretation but restrict the object program so that its semantics is preserved in the meta-program. This approach has been explored by Martens and De Schreye (1992b), (1992a) using the concept of *language independence*. (Informally, a program is language independent when the perfect Herbrand models are not affected by the addition of new constant and function symbols.) In the next subsection, we examine how each of these approaches may be used to prove the correctness of the definition *Solve* with respect to the intended semantics.

2.2 Reflective Predicates

By representing variables as variables, the non-ground representation provides implicit support for the reflection of the semantics of unification. For example, a meta-program may define a reflective predicate *Unify* by the fact

$Unify(u, u)$

so that the goal

$\leftarrow Unify(\right.$
 $\quad Member(x, Cons(x, y)),$
 $\quad \left. Member(1, Cons(z, Cons(2, Cons(3, Nil)))) \right)$

will succeed with

$x = 1, y = Cons(2, Cons(3, Nil)), z = 1.$

The ability to use the underlying unification mechanism for both the object program and its representation makes it easy to define reflective predicates whose semantics in the meta-program correspond to the semantics of the object program.

The meta-interpreter \mathbf{V} in Figure 3 assumes that the object program is a normal logic program. It defines a reflective predicate *Solve/1*. Given a normal object program P , the program \mathbf{V}_P consists of the program \mathbf{V} together with a set of facts defining *Clause/2* and representing P . For example, let P be the program in Figure 2, and $\leftarrow G$ the goal

$\leftarrow \neg Member(x, Cons(2, (Cons(3, Nil)))) \wedge$
 $\quad Member(x, Cons(1, Cons(2, (Cons(3, Nil)))))$.

Then both the goal $\leftarrow G$ and the goal $\leftarrow Solve([G])$

$$\begin{aligned}
& \text{Solve}(\text{True}) \\
\text{Solve}((a \text{ And } b)) & \leftarrow \\
& \quad \text{Solve}(a) \wedge \\
& \quad \text{Solve}(b) \\
\text{Solve}(\text{Not } a) & \leftarrow \\
& \quad \neg \text{Solve}(a) \\
\text{Solve}(a) & \leftarrow \\
& \quad \text{Clause}(a, b) \wedge \\
& \quad \text{Solve}(b)
\end{aligned}$$

Fig. 3. The Vanilla meta-interpreter \mathbf{V}

$$\leftarrow \text{Solve}(
\begin{aligned}
& \text{Not Member}(x, \text{Cons}(2, (\text{Cons}(3, \text{Nil})))) \text{ And} \\
& \text{Member}(x, \text{Cons}(1, \text{Cons}(2, (\text{Cons}(3, \text{Nil}))))))
\end{aligned}
)$$

have the computed answer

$x = 1.$

The predicate $\text{Solve}/1$ is intended to satisfy the following reflective principles which are adaptations of those given in Subsection 1.4.

$$P \vdash_{\mathcal{L}} Q \text{ iff } \mathbf{V}_P \vdash_{\mathcal{M}} \text{Solve}([Q])$$

$$\text{comp}(P) \models_{\mathcal{L}} Q \text{ iff } \text{comp}(\mathbf{V}_P) \models_{\mathcal{M}} \text{Solve}([Q])$$

Here Q is assumed to be a conjunction of ground literals. It has been shown that the relation $\text{Solve}/1$ has the intended semantics in program \mathbf{V}_P if either the interpretation of \mathbf{V}_P is typed or the object program represented by the definition of $\text{Clause}/2$ is language independent. Each of these conditions is described in turn.

We first consider using a typed interpretation of \mathbf{V}_P . There are (at least) two types in the interpretation, o and μ , where o is intended for object language terms and μ for object language formulas. The representation for the terms and formulas in the object language together with the intended types for their interpretation is as follows.

Object Symbols	Meta Symbols	Type
Constant	Constant	o
Function of arity n	Function of arity n	$o * \dots * o \rightarrow o$
Proposition	Constant	μ
Predicate of arity n	Function of arity n	$o * \dots * o \rightarrow \mu$
Binary connective	Function of arity 2	$\mu * \mu \rightarrow \mu$
Unary connective	Function of arity 1	$\mu \rightarrow \mu$

In addition, the predicates *Solve/1* and *Clause/2* have each of their arguments of type μ . The following result is proved in (Hill & Lloyd 1989).

Theorem 2.2.1. *Let P be a normal program and $\leftarrow Q$ a normal goal. Let \mathbf{V}_P be the program defined above. Then the following hold.*

1. *$\text{comp}(P)$ is consistent iff $\text{comp}(\mathbf{V}_P)$ is consistent.*
2. *θ is a correct answer for $\text{comp}(P) \cup \{\leftarrow Q\}$ iff θ is a correct answer for $\text{comp}(\mathbf{V}_P) \cup \{\leftarrow \text{Solve}(Q)\}$.*
3. *$\neg Q$ is a logical consequence of $\text{comp}(P)$ iff $\neg \text{Solve}(Q)$ is a logical consequence of $\text{comp}(\mathbf{V}_P)$.*

Theorem 2.2.2. *Let P be a normal program and $\leftarrow Q$ a normal goal. Let \mathbf{V}_P be the program defined above. Then the following hold.*

1. *θ is a computed answer for $P \cup \{\leftarrow Q\}$ iff θ is a computed answer for $\mathbf{V}_P \cup \{\leftarrow \text{Solve}(Q)\}$.*
2. *$P \cup \{\leftarrow Q\}$ has a finitely failed SLDNF-tree iff $\mathbf{V}_P \cup \{\leftarrow \text{Solve}(Q)\}$ has a finitely failed SLDNF-tree.*

It is important to note here that, although the declarative semantics of \mathbf{V}_P requires a model which is typed, the procedural semantics for the program P and for \mathbf{V}_P is the same and that, apart from checking that the given goal is correctly typed, no run-time type checking is involved.

Martens and De Schreye have provided an alternative solution to the semantics of \mathbf{V}_P that does not require the use of types. We now give a summary of the semantics that they have proposed. Their work requires the object programs to be stratified and satisfy the condition of language independence.

Definition 2.2.3. *Let P be a program. A language for P ¹ is any language \mathcal{L} such that each clause in P is a well-formed expression in the language \mathcal{L} .*

Let \mathcal{L}_P be the language defined using just the symbols occurring in P . Then any language for P will be an extension of \mathcal{L}_P .

Definition 2.2.4. *Let P be a stratified program. Then P is said to be language independent if the perfect Herbrand model of P is independent of the choice of language for P .*

Language independence is an undecidable property. Thus it is important to find a subclass of the language independent programs that can be recognised syntactically. The following well-known concept of range restriction determines such a class.

Definition 2.2.5. *A clause in a program P is called range restricted if every variable in the clause appears in a positive literal in the body of*

¹Note that this definition of a language of a program just applies to the discussion of Martens and De Schreye's results and does not hold in the rest of this chapter.

the clause. A program is called *range restricted* if all its clauses are range restricted.

It is shown in (Martens & De Schreye 1992b) that the set of range restricted stratified programs is a proper subset of the language independent programs. It is also demonstrated that, if P is a definite program and G a definite goal, P is language independent if and only if all computed answers (using SLD-resolution) for $P \cup \{G\}$ are ground.

The correctness of the \mathbf{V}_P program is stated in terms of the perfect model for the object program and the weakly perfect model for the meta-program. It is shown that if the object program P is stratified, then the program \mathbf{V}_P has a weakly perfect model. Note that a stratified program always has a perfect model.²

We can now state the main result.

Theorem 2.2.6. *Let P be a stratified language independent normal program and \mathbf{V}_P the meta-program, as defined above. Then, for every predicate p/n occurring in P the following hold.*

1. *If t_1, \dots, t_n are ground terms in the language defined by the symbols occurring in \mathbf{V}_P , then $\text{Solve}(p(t_1, \dots, t_n))$ is true in the weakly perfect Herbrand model of \mathbf{V}_P iff $p(t_1, \dots, t_n)$ is true in the perfect Herbrand model of P .*
2. *If t_1, \dots, t_n are ground terms in the language defined by the symbols occurring in P , then $\text{Solve}(\text{Not } p(t_1, \dots, t_n))$ is true in the weakly perfect Herbrand model of \mathbf{V}_P iff $p(t_1, \dots, t_n)$ is not true in the perfect Herbrand model of P .*

The main issue distinguishing the typed and language independent approaches is the criterion that is used in determining the language of a program. Either the language of a program is inferred from the symbols it contains or the language can be defined explicitly by declaring the symbols. As the language of the program \mathbf{V}_P must include a representation of all the symbols of the object program, it is clear that if we require the language of a program to be explicitly declared, the meta-program will need to distinguish between the terms that represent object terms and those that represent object formulas. This of course leads naturally to a typed interpretation. On the other hand, if the language of a program is fixed by the symbols actually appearing in that program, then we need to ensure the interpretation is unchanged when the program is extended with new symbols. This leads us to look at the concept of language independence which is the basis of the second approach.

²A definition of a weakly perfect model is in (Martens & De Schreye 1992a). A definition of a stratified program can be found in (Lloyd 1987) as well as in (Martens & De Schreye 1992a).

```

PSolve(True, True)
PSolve(x And y, xproof And yproof) ←
    PSolve(x, xproof) ∧
    PSolve(y, yproof)
PSolve(Not x, True) ←
    ¬PSolve(x, _)
PSolve(x, x If yproof) ←
    Clause(x, y) ∧
    PSolve(y, yproof)

```

Fig. 4. The Proof-Tree meta-interpreter **W**

The advantages of using a typed interpretation is that no extra conditions are placed on the object program. The usual procedural semantics for logic programming can be used for both typed and untyped programs. Thus the type information can be omitted from the program although it seems desirable that the intended types of the symbols be indicated at least as a comment to the program code. A possible disadvantage is that we must use many-sorted logic to explain the semantics of the meta-program instead of the better known unsorted logic. The alternative of coding the type information explicitly as part of the program is not desirable since this would create an unnecessary overhead and adversely affect the efficiency of the meta-interpreter.

The advantage of the language independence approach is that for definite language independent object programs, the semantics of the **V** program is based on that of unsorted first order logic. However, many common programs such as the program in Figure 2 are not language independent. Moreover, as soon as we allow negation in the bodies of clauses, any advantage is lost. Not only do we still require the language independence condition for the object program, but we can only compare the weakly perfect model of the meta-program with the perfect model of the object program.

We conclude this subsection by presenting in Figure 4 an extended form of the program **V** in Figure 3. This program, which is a typical example of what the non-ground interpretation can be used for is adapted from (Sterling & Shapiro 1986). The program **W** defines a predicate *PSolve/2*. The first argument of *PSolve/2* corresponds to the single argument of *Solve* but the second argument must be bound to the program's proof of the first argument.

Given a normal object program *P*, the program **W_P** consists of the program **W** together with a set of facts defining *Clause/2* and representing *P*. If *P* is the program in Figure 2, then the goal

```

← PSolve(
    Not Member(x, Cons(2, (Cons(3, Nil)))) And
    Member(x, Cons(1, Cons(2, (Cons(3, Nil))))),
    proof)

```

has the computed answer

```

x = 1
proof = True And (Member(1, Cons(1, Cons(2, Cons(3, Nil)))) If True).

```

2.3 Meta-Programming in Prolog

The language Prolog has a number of built-in predicates that are useful for meta-programming. These can be divided into two categories. Those with a declarative semantics and those whose semantics is purely procedural.

The first category includes `functor/3`, `arg/3`, and `=../2`. `functor` is true if the first argument is a term, the second is the name of the top-level function for this term and the third is its arity. `arg` is true if the first argument is a positive integer i , the second, of the form $f(t_1, \dots, t_n)$ ($n \geq i$), and the third, the i th argument t_i . For a Prolog meta-program containing the non-ground representation of the Chase program in Figure 1, the definition of the system predicates `functor` and `arg` would be:

```

functor(tom, tom, 0).
functor(jerry, jerry, 0).
functor(cat(_), cat, 1).
functor(mouse(_), mouse, 1).
functor(chase(_, _), chase, 2).

```

```

arg(1, cat(X), X).
arg(1, mouse(X), X).
arg(1, chase(X, Y), X).
arg(2, chase(X, Y), Y).

```

The predicate `=..` is a binary infix predicate which is true when the left hand argument is a term of the form f or $f(t_1, \dots, t_n)$ and the right hand argument is the list $[f, t_1, \dots, t_n]$. This predicate is not really necessary and can be defined in terms of `functor` and `arg`.

```

Term =.. [Function| Args] :-
    functor(Term, Function, Arity),
    findargs(Arity, Args, Term).

```

```

findargs(0, [], _).
findargs(Argno, [Arg|Args], Term) :-
    arg(Argno, Term, Arg),
    Argno1 is Argno - 1,
    findargs(Argno1, Args Term).

```

The second category includes meta-logical predicates such as `var/1`, `nonvar/1`, and `atomic/1` and the dynamic predicates such as `assert/1` and `retract/1`.

The predicate `var/1` tests whether its argument is currently uninstantiated, while `nonvar/1` is the opposite of `var` and tests whether its only argument is currently instantiated. `atomic/1` succeeds if its argument is currently instantiated to a constant. These predicates are not declarative since computed answers are not correct answers. Consider the following Prolog goals.

```
?- var(X)
?- var(3)
```

The first goal succeeds with no binding, suggesting that `var(X)` is true for all instances of `X`, while the second goal fails.

The predicates `assert/1` and `retract/1` allow modification of the program being interpreted. On execution of `assert(t)`, provided `t` is in the correct syntactical form for a clause, the current instance of `t` is added as a clause to the program. `retract` is the opposite of `assert`. On execution of `retract(t)`, if there is a clause in the current program that unifies with `t`, then the first such clause will be removed from the program. Since these modify the program being executed and their effect is not undone on backtracking, it is clear that they do not have a declarative semantics.

3 The Ground Representation

In this section we review the work that has been done in logic programming on meta-programming using the ground representation, that is, where any expression of the object program is represented by a ground expression in the meta-program. As in the previous section, we begin by outlining the historical background to the use of this representation.

Use of a ground representation in logic can be traced back to the work of Gödel, who gave a method (called a Gödel numbering) for representing expressions in a first order language as natural numbers (Gödel 1931). This was defined by giving each symbol of the logic a unique positive odd number, and then, using the sequence of symbols in a logical expression, a method for computing a unique number for that expression. It was not only possible to compute the representation of an expression, but also, given such a number, determine the unique expression that it represented. Using the properties of the natural numbers, this representation has been used in proving properties of the logic. Feferman (1962), who applied Gödel's ideas to meta-mathematics, introduced the concept of a reflective principle.

Weyhrauch (1980) designed a proof checker, called FOL which has special support for expressing properties of a FOL structure (using a ground representation) in a FOL meta-theory. An important concept in FOL is the simulation structure. This is used in the meta-reasoning part of FOL

for reflective axioms that define an intended interpretation of the representation of the object language in terms of other theories already defined in FOL (or in LISP). A FOL meta-theory has, as simulation structure, the object theory and the mappings between the language of the object theory and the language of the meta-theory. Weyhrauch generalised the reflection principle (as defined by Feferman) to be defined simply as a statement of a relation between a theory and its meta-theory. He gives, as an example, a statement of the correspondence between provability of a formula f in the object theory and a predicate $Pr/2$ in the metatheory. $Pr/2$ is intended to be true if the first argument represents the proof of f and the second argument represents f .

Bowen and Kowalski (1982) showed how the idea of using a representation similar to that used in FOL could be adapted for a logic programming system. The actual representation is not specified although a simple syntactic scheme is used in the paper whereby a symbol " $P(x, Bill)$ " is used to denote a term of the meta-language which names $P(x, Bill)$. In addition, where a symbol, say A , is used to denote a term or formula of the object language, A' denotes the term in the meta-language representing A . In fact, the key point of this paper was not to give a detailed scheme for the representation of expressions in the object language, but to represent the provability relation $\vdash_{\mathcal{L}}$ for the object program with language \mathcal{L} by means of a predicate $Demo/2$ in the meta-program with language \mathcal{M} . Thus, in the context of a set of sentences Pr of \mathcal{M} , $Demo$ represents $\vdash_{\mathcal{L}}$ if and only if, for all finite sets of sentences A and single sentences B of \mathcal{L} ,

$$A \vdash_{\mathcal{L}} B \text{ iff } Pr \vdash_{\mathcal{M}} Demo(A', B').$$

It was not intended that $\neg Demo$ in Pr should represent unprovability since provability is semi-decidable.

The ideas of Bowen and Kowalski were demonstrated by Bowen and Weinberg (1985) using a logic programming system called MetaProlog, which extended Prolog with special meta-programming facilities. These consisted of three system predicates, `demo/3`, `add_to/3`, and `drop_from/3`. The first of these corresponded to the $Demo$ predicate described by Bowen and Kowalski. The predicate `demo/3` defines a relation between the representations of an object program P , a goal G , and a proof R . `demo` is intended to be true when G is a logical consequence of P and a proof of this is given by R . Predicates `add_to/3`, and `drop_from/3` are relations between the representations of a program P , a clause C , and another program Q . `add_to/3` and `drop_from/3` are intended to be true when Q can be obtained from P by, respectively, adding or deleting the clause C . These predicates provided only the basic support for meta-programming and, to perform other meta-programming tasks, a meta-program has to use the non-declarative predicates in the underlying Prolog. Thus, although the

theoretical work is based on a ground representation, meta-programming applications implemented in MetaProlog are often forced to use the non-ground representation.

Apart from the work of Bowen and Weinberg, meta-programming using the ground representation remained a mainly theoretical experiment for a number of years. Stimulation for furthering the research on this subject was brought about by the initiation in 1988 of a biennial series of workshops on meta-programming in logic. A paper (Hill & Lloyd 1989) in the 1988 workshop built upon the ideas of Bowen and Kowalski and described a more general framework for defining and using the ground representation. In this representation, object clauses were represented as facts in the meta-program and did not allow for changing the object program. Subsequent work modified this approach so as to allow for more dynamic meta-programming (Hill & Lloyd 1988). Gödel is a programming system based on these ideas (Hill & Lloyd 1994). This system has considerable specialist support for meta-programming using a ground representation. Those aspects of Gödel, pertinent to the meta-programming facilities, are described in Subsection 3.3.

3.1 The Representation

In Section 2, a simple scheme for representing a first order language based on that employed by Prolog was described. However, for most meta-programs such a representation is inadequate. Meta-programs often have to reason about the computational behaviour of the object program and, for this, only the ground representation is suitable. A program can be viewed from many angles; for example, the language, the order of statements, the modular structure. Meta-programs may need to reason about any of these. In this subsection, we first consider the components that might constitute a program and how they may be represented.

In order to discuss the details of how components of an object program may be represented as terms in a meta-program, we need to understand the structure of the object program that is to be represented. Object programs are normally parsed at a number of structural levels, ranging from the individual characters to the modules (if the program is modular) that combine together to form the complete program. We assume here that an object program has some of the following structural levels.

1. Character
2. Symbol
3. Language element
4. Statement or declaration
5. Module
6. Program

For example, for a normal logic program (with a modular structure) these

would correspond to the following.

1. Alphabetic and non-alphabetic character
2. Constant, function, proposition, predicate, connective, and variable
3. Term and formula
4. Clause
5. Set of predicate definitions
6. Logic program

Note that levels 1, 2, and 3 contribute to the language of a program, while levels 4, 5, and 6 are required for the program's theory. At each structural level a number of different kinds of tree structure (called here a *unit*) are defined. That is, a unit at a structural level other than the lowest will be a tree whose nodes are units defined at lower levels. At the lowest structural level, the units are the pre-defined characters allowed by the underlying programming system. Thus a subset of the trees whose nodes are units defined at or below a certain structural level will form the set of units at the next higher level. For example, in the program in Figure 2, M and $<$ are characters, $Member$ and x , are symbols, and $Member(x, Cons(x, y))$ is a formula. Note that, as a tree may consist of just a single unit, some units may occur at more than one level. Thus the character x is also a variable as well as a term of the language. A representation will be defined for one or more of these structural levels of the object program. Moreover, it is usual for the representation to distinguish between these levels, so that, for example, a constant such as A may have three representations depending on whether it is being viewed as a character, constant, or term. Note that this provision of more than one representation for the different structural levels of a language is usual in logic. In particular, the arithmetization of first order logic given by the Gödel numbering has a different number for a constant when viewed as a symbol to that when it is viewed as a term.

The actual structural levels that may be represented depend on the tasks the meta-program wishes to perform. In particular, if we wish to define a reflective predicate that has an intended interpretation of SLD-resolution, we need to define unification. For this, a representation of the symbols is required. If we wish to be able to change the object program using new symbols created dynamically, then a representation of the characters would be needed.

A representation is in fact a coding and, as is required for any cipher, a representation should be invertible (injective) so that results at the meta-level can be interpreted at the object-level³. Such an inverse mapping we call here a *re-representation*. To ensure that not only the syntax and

³There is of course an alternative explanation of why a representation must be injective: it is the inverse of denotation. Since denotation must be functional (any term has (at most) one denotation), the inverse of denotation must be injective.

structural level of the object element can be recovered but also the kind of unit at that level (for example, if it is a language element in a logic program, whether it is a term, atom, or formula) that is represented, the representation has to include this information as part of the coding.

We give, as an example, a simple ground representation for a normal program. This is used in the next subsection by the Instance-Demo program in Figure 5 and the SLD-Demo program in Figure 6. First we define the naming relation for each symbol in the language and then define the representation for the terms and formulas (using just the connectives \wedge, \leftarrow, \neg) that can be constructed from them. Individual characters are not represented. For this representation, it is assumed that the language of the meta-program includes three types: the type s for terms representing object symbols; the type o for terms representing object terms; and the type μ for terms representing object formulas. It is also assumed that the language includes the type $List(a)$ for any type a together with the usual list constructor $Cons$ and constant Nil . The language must also include the functions $Term/2, Atom/2, V/1, C/1, F/1, P/1, And/2, If/2,$ and $Not/1$ whose intended types are as follows.

Function	Type
$Term$	$s * List(o) \rightarrow o$
$Atom$	$s * List(o) \rightarrow \mu$
V	$Integer \rightarrow o$
C	$Integer \rightarrow s$
F	$Integer \rightarrow s$
P	$Integer \rightarrow s$
And	$\mu * \mu \rightarrow \mu$
If	$\mu * \mu \rightarrow \mu$
Not	$\mu \rightarrow \mu$

The representation of the variables, non-logical symbols, and connectives is as follows.

Object Symbol	Representation
Variable	Term $V(n)$ $n \in Integer$
Constant	Term $C(i)$ $i \in Integer$
Function	Term $F(j)$ $j \in Integer$
Proposition/ Predicate	Term $P(k)$ $k \in Integer$
\wedge	Function $And/2$
\leftarrow	Function $If/2$
\neg	Function $Not/1$

To give the representation of the terms and formulas, suppose D is a

constant represented by $C(i)$, G/n a function represented by $F(j)$, and Q/n a predicate represented by $P(k)$.

Object Expression		Representation
Constant	D	Term $Term(C(i), [])$
Term	$G(t_1, \dots, t_n)$	Term $Term(F(j), [[t_1], \dots, [t_n]])$
Atom	$Q(t_1, \dots, t_n)$	Term $Term(P(k), [[t_1], \dots, [t_n]])$
Formula	$A \wedge B$	Term $And([A], [B])$
Formula	$\neg A$	Term $Not([A])$
Formula	$A \leftarrow B$	Term $If([A], [B])$

As an example of this representation, consider the language of the Member program in Figure 2.

Object Symbol	Representation
x	$V(0)$
y	$V(1)$
z	$V(2)$
Nil	$C(0)$
$i > 0$	$C(i)$
$i \leq 0$	$C(i - 1)$
$Cons$	$F(0)$
$Member$	$P(0)$

Thus the atom

$Member(x, [1, 2])$

is represented by the term

$Atom(P(0),$
 $[V(0),$
 $Term($
 $F(0),$
 $[Term(C(1), []),$
 $Term(F(0), [Term(C(2), []), Term(C(0), [])])])$
 $].$

The representation of a program clause which is a fact

H

is

$If([H], True)$

The representation of a program clause

$H \leftarrow B$

is

$If([H], [B])$

which is the same as the formula representation of $H \leftarrow B$.

The theory of a program is represented as a list of clauses. Using this representation, the Member program is represented by the following list.

$$\begin{aligned} [& \text{If}(\text{Atom}(P(0), [V(0), \text{Term}(F(0), [V(0), V(1)])]), \text{True}), \\ & \text{If}(\text{Atom}(P(0), [V(0), \text{Term}(F(0), [V(1), V(2)])]), \\ & \quad \text{Atom}(P(0), [V(0), V(2)]) \quad] \end{aligned}$$

This representation is defined for any normal programs and is used for many of the examples in this chapter. However, the main limitation with this representation is that there is no representation of the characters making up the object symbols. Thus this representation does not facilitate the generation of new object languages. In addition, it has been assumed that the object program has no module structure. By representing a module as a list of clauses and a program as a list of modules, a meta-program could reason about the structure of a modular program.

It can be seen that a representation such as the one described above that encodes structural information, is difficult to use and prone to user errors. An alternative representation is a string. A string, which is a finite sequence of characters, is usually indicated by enclosing the sequence in quotation marks. For example, "ABC" denotes the string A, B, C . With the string representation, the atom

$\text{Member}(x, [1, 2])$

is represented as

"Member(x, [1, 2])"

If a string concatenation function ++ is available, then unspecified components can be expressed using variables in the meta-program. For example

"Member(" ++ x ++ ", [1, 2])"

defines a term that corresponds, using the previous representation to the (non-ground) term

$$\begin{aligned} \text{Atom}(\text{P}(0), \\ [\text{x}, \\ \quad \text{Term}(\\ \quad \quad \text{F}(0), \\ \quad \quad [\text{Term}(\text{C}(1), []), \\ \quad \quad \quad \text{Term}(\text{F}(0), [\text{Term}(\text{C}(2), []), \text{Term}(\text{C}(0), [])])] \\ \quad]). \end{aligned}$$

As the string representation carries no structural information, this representation is computationally very inefficient. Thus, it is desirable for a representation such as a string as well as one that is more structurally descriptive to be available. The meta-programming system should then provide a means of converting from one representation to the other.

Most researchers on meta-programming define a specific representation

suitable for their purposes without explaining why the particular representation was chosen. However, Van Harmelen (1992) has considered the many different ways in which a representation may be defined and how this may assist in reasoning about the object theory. We conclude this subsection with interesting examples of non-standard representations based on examples in his paper.

The first example defines a representation that encodes information about the degree of instantiation of formulae in the object language. The object theory encodes graphs as facts of the form $Edge(N_i, N_j)$ together with a rule defining the transitive closure.

$$Connected(n_1, n_2) \leftarrow Edge(n_1, n_3) \wedge Connected(n_3, n_2)$$

By choice of representation, we can construct a meta-theory containing the control assertion that the conjunction defining *Connected* should be executed from left to right if the first argument of *Connected* is given, but from right to left if the second argument is given. The representation assigns different terms to the object formulas, depending on their degree of instantiation.

Object Formula	Representation
$Edge(N15, x)$	$GroundVar(Edge, N15, V(1))$
$Connected(N15, x)$	$GroundVar(Connected, N15, V(1))$
$Edge(x, N15)$	$VarGround(Edge, V(1), N15')$
$Connected(N15, x)$	$VarGround(Connected, V(1), N15)$
$Edge(x, y)$	$VarVar(Edge, V(1), V(2))$
$Connected(N15, N16)$	$GroundGround(Connected, N15, N16)$

With this representation we can define the atoms that may be selected by means of a predicate *Selectable* in the meta-program.

$$Selectable(GroundGround(-, -, -))$$

$$Selectable(GroundVar(-, -, -))$$

$$Selectable(VarGround(Edge, -, -))$$

This would be taken into account by an interpreter of this program included in the meta-program of the object program.

The second example is taken from the field of knowledge-based systems and concerns the representation of the difference between implication when used to mean a causation and when used as a specialization. Such implications may require different inference procedures. The representation could be as follows.⁴

Object Formula	Representation
$AcuteMenin \rightarrow Menin$	$TypeOf(AcuteMenin, Menin)$
$Meningococcus \rightarrow Menin$	$Causes(Meningococcus, Menin)$

⁴*Menin* and *AcuteMenin* are abbreviations of *Meningitis* and *Acute Meningitis*, respectively.

The meta-program can then specify the inference steps that would be appropriate for each of these relations.

Although the idea of being able to define a representation tailored for a particular application is attractive, no programming system has been implemented that provides support for this.

3.2 Reflective Predicates

With the ground representation, the reflective predicates have to be explicitly defined. For example, a predicate *GroundUnify/2* that is intended to be true if its arguments represent identical terms in the object language, must be defined so that its completed definition corresponds to the Clark equality theory. Such a definition requires a full analysis of the terms representing object terms and atoms. This is computationally expensive compared with using *Unify/2* to unify expressions in the non-ground representation (see Subsection 2.2).

There are two basic styles of meta-interpreter that use the ground representation and have been discussed in the literature. The first style is derived from an idea proposed in (Kowalski 1990) and has a similar form to the program **V** in Figure 3 but uses the ground rather than the non-ground representation. In this subsection, we present, in Figure 5, an interpreter **I** based on this proposal. This style of meta-interpreter and similar meta-programs are being used in several programs although a complete version and a discussion of its semantics has not previously been published. In the second style the procedural semantics of the object program is intended to be a model of the meta-interpreter. For example, the meta-interpreter outlined in (Bowen & Kowalski 1982) is intended to define SLD-resolution. An extension of this meta-interpreter for SLDNF-resolution is shown in (Hill & Lloyd 1989) to be correct with respect to its intended interpretation. An outline of such a meta-interpreter **J** is given in Figure 6. The Gödel program SLD-Demo, also based on this style, is presented in the next subsection in Figure 9.

Both the programs **I** and **J** make use of the ground representation defined in the previous subsection. These programs also require an additional type σ to be used for the bindings in a substitution. The function *Bind/2* is used to construct such a binding. This has domain type *Integer * o* and range type σ .

The meta-interpreter **I** in Figure 5 defines the predicate *IDemo/3*. This program is intended to satisfy the reflective principles

$$P \vdash_{\mathcal{L}} Q\theta \text{ iff } \mathbf{I} \vdash_{\mathcal{M}} IDemo([P], [Q], [Q\theta])$$

$$\text{comp}(P) \models_{\mathcal{L}} Q\theta \text{ iff } \text{comp}(\mathbf{I}) \models_{\mathcal{M}} IDemo([P], [Q], [Q\theta]).$$

Here, Q denotes a conjunction of literals and θ a substitution that grounds Q . These reflective principles, which are adaptations of those given in Sub-

section 1.4, ensure that provability and logical consequence for the object program are defined in the meta-program.

The types of the predicates in Program **I** are as follows.

Predicate	Type
<i>IDemo</i>	$List(\mu) * \mu * \mu$
<i>IDemo1</i>	$List(\mu) * \mu$
<i>InstanceOf</i>	$\mu * List(\mu)$
<i>InstFormula</i>	$\mu * \mu * List(\sigma) * List(\sigma)$
<i>InstTerm</i>	$o * o * List(\sigma) * List(\sigma)$
<i>InstArgs</i>	$List(o) * List(o) * List(\sigma) * List(\sigma)$

The above reflective principles intended for program **I** are similar to those intended for the meta-interpreter \mathbf{V}_P . Apart from the representation of the object variables, the main difference between these programs is that \mathbf{V}_P includes the representation of the object program P . Thus, to make it easier to compare the programs **V** and **I**, the first clause of **I** needs to be replaced by

$$\begin{aligned}
 IDemo(p, x, y) \leftarrow & \\
 & ObjectProgram(p) \wedge \\
 & InstanceOf(x, y) \wedge \\
 & IDemo1(p, y)
 \end{aligned}$$

and then define the program \mathbf{I}_P to consist of this modified form of **I** together with a unit clause

$$ObjectProgram([P])$$

We expect that similar results to those of Theorems 2.2.1 and 2.2.2 will then apply to \mathbf{I}_P . Further research is needed to clarify the semantics of the program **I** in Figure 5 and the variation \mathbf{I}_P described above.

The meta-interpreter **J** in Figure 6 defines the predicate $JDemo/3$. This program is intended to satisfy the reflective principle

$$P \vdash_{\mathcal{L}} Q\theta \text{ iff } \text{comp}(\mathbf{I}) \models_{\mathcal{M}} JDemo([P], [Q], [Q\theta]).$$

Here, Q denotes a conjunction of literals and θ a substitution that grounds Q .

The types of the predicates in **J** are as follows.

$$\begin{aligned}
IDemo(p, x, y) &\leftarrow \\
&\quad InstanceOf(x, y) \wedge \\
&\quad IDemo1(p, y) \\
\\
IDemo1(-, True) &. \\
IDemo1(p, And(x, y)) &\leftarrow \\
&\quad IDemo1(p, x) \wedge \\
&\quad IDemo1(p, y) \\
IDemo1(p, Not(x)) &\leftarrow \\
&\quad \neg IDemo1(p, x) \\
IDemo1(p, Atom(P(k), xs)) &\leftarrow \\
&\quad Member(z, p) \wedge InstanceOf(z, If(Atom(P(k), xs), b)) \wedge \\
&\quad IDemo1(p, b) \\
\\
InstanceOf(x, y) &\leftarrow InstFormula(x, y, [], -) \\
\\
InstFormula(Atom(P(k), xs), Atom(P(k), ys), s, s1) &\leftarrow \\
&\quad InstArgs(xs, ys, s, s1) \\
InstFormula(And(x, y), And(z, w), s, s2) &\leftarrow \\
&\quad InstFormula(x, z, s, s1) \wedge \\
&\quad InstFormula(y, w, s1, s2) \\
InstFormula(If(x, y), If(z, w), s, s2) &\leftarrow \\
&\quad InstFormula(x, z, s, s1) \wedge \\
&\quad InstFormula(y, w, s1, s2) \\
InstFormula(Not(x), Not(z), s, s1) &\leftarrow \\
&\quad InstFormula(x, z, s, s1) \\
InstFormula(True, True, s, s) & \\
\\
InstTerm(V(n), x, [], [Bind(n, x)]) & \\
InstTerm(V(n), x, [Bind(n, x)|s], [Bind(n, x)|s]) & \\
InstTerm(V(n), x, [Bind(m, y)|s], [Bind(m, y)|s1]) &\leftarrow \\
&\quad n \neq m \wedge \\
&\quad InstTerm(V(n), x, s, s1) \\
InstTerm(C(i), C(i), s, s) & \\
InstTerm(Term(F(j), xs), Term(F(j), ys), s, s1) &\leftarrow \\
&\quad InstArgs(xs, ys, s, s1) \\
\\
InstArgs([], [], s, s) & \\
InstArgs([x|xs], [y|ys], s, s2) &\leftarrow \\
&\quad InstTerm(x, y, s, s1) \wedge \\
&\quad InstArgs(xs, ys, s1, s2)
\end{aligned}$$

Fig. 5. The Instance-Demo program I

```

JDemo(p, q, q1) ←
  MaxForm(q, n) ∧
  Derivation(p, q, True, [], s, n) ∧
  ApplyToForm(s, q, q1)

Derivation(_, q, r, s, s, _)
Derivation(p, q, r, s, t, n) ←
  SelectLit(Atom(P(k), xs), q) ∧
  Member(If(Atom(P(k), ys), ls), p) ∧
  Resolve(q, Atom(P(k), xs), If(Atom(P(k), ys), ls), s, s1, r1, n, n1) ∧
  Derivation(p, r1, r, s1, t, n1)

Derivation(p, q, r, s, s, _) ←
  SelectLit(Not(a), q) ∧
  ApplyToForm(s, a, a1) ∧
  Ground(a1) ∧
  ¬Derivation(p, a1, True, [], 0) ∧
  ReplaceConj(q, Not(a), True, r)

Resolve(q, Atom(P(k), xs), If(Atom(P(k), ys), ls), s, t, r, m, n) ←
  Rename(m, If(Atom(P(k), ys), ls), If(Atom(P(k), y1s), l1s), n) ∧
  UnifyTerms(y1s, xs, s, t) ∧
  ReplaceConj(q, Atom(P(k), xs), l1s, r)

```

Fig. 6. The SLD-Demo program J

Predicate	Type
<i>JDemo</i>	$List(\mu) * \mu * \mu$
<i>Derivation</i>	$List(\mu) * \mu * \mu * List(\sigma) * List(\sigma) * Integer$
<i>Resolve</i>	$\mu * \mu * \mu * List(\sigma) * List(\sigma) * \mu * Integer * Integer$
<i>MaxForm</i>	$\mu * Integer$
<i>SelectLit</i>	$\mu * \mu$
<i>Ground</i>	μ
<i>ReplaceConj</i>	$\mu * \mu * \mu * \mu$
<i>Rename</i>	$Integer * \mu * \mu * Integer$
<i>UnifyTerms</i>	$List(o) * List(o) * List(\sigma) * List(\sigma)$
<i>ApplyToForm</i>	$List(\sigma) * \mu * \mu$

It is intended that the first argument of *Derivation*/6 represents a normal program *P*, the second and third represent conjunctions of literals *Q* and *R*, the fourth and fifth represent substitutions θ and ϕ , and the sixth is an index *n* for renaming variables. *Derivation*/6 is true if there is an SLD-derivation of $P \cup \{\leftarrow Q\theta\}$ ending in the goal $\leftarrow R\phi$. The predicate *MaxForm*/2 finds the maximum of the variable indices (that is, *n* in $V(n)$)

in the representation of a formula; *SelectLit/2* selects a literal from the body of a clause; *Ground/1* checks that a formula is ground; *Rename/4* finds a variant of the formula in the second argument by adding the number in the first argument to the index of each of its variables and setting the last argument to the maximum of the new variable indices; *Resolve* performs a single derivation step; *ReplaceConj/4* removes an element from a conjunction and replaces it by another literal or conjunction of literals; *ApplyToForm/3* applies a substitution to a formula; and *UnifyTerms/4* finds an mgu for two lists of terms obtained by applying the substitution in the third argument to the lists of terms in the first and second arguments.

Programs **I** and **J** can be used with the Member program in Figure 2 as their object program. A goal for **I** or **J** that represents the object-level query

$\leftarrow \text{Member}(x, [1, 2])$

is given in Figure 7 (where *Demo* denotes either *IDemo* or *JDemo*). Figure 7 also contains the computed answers for this goal using the programs in Figures 5 and 6.

The goal in Figure 7 has a ground term representing the object program. It has been observed that if a meta-interpreter such as **I** or **J** was used by a goal as above, but where the third argument representing the object program was only partly instantiated, then a program that satisfied the goal could be created dynamically. However, in practice, such a goal would normally flounder if SLDNF-resolution was used as the procedural semantics. Christiansen (1994) is investigating the use of constraint techniques in the implementation of the meta-interpreter so as to avoid this problem. These techniques have the potential to implement various forms of reasoning including abduction and inductive logic programming.

The Instance-Demo program **I** relies on using the underlying procedures for unification and standardising apart. Thus it would be difficult to adapt the program to define complex computation rules that involve non-trivial co-routining. However, the SLD-Demo program **J**, can easily be modified to allow for arbitrary control strategies.

3.3 The Language Gödel and Meta-Programming

The Gödel language (Hill & Lloyd 1994) is a logic programming language that facilitates meta-programming using the ground representation. The provision is mainly focussed on the case where the object program is another Gödel program, although there are also some basic facilities for representing and manipulating expressions in any structured language.

One of the first problems encountered when using the ground representation is how to obtain a representation of a given object program and goals for that program. It can be seen from the example in Figure 7, that a ground representation of even a simple formula can be quite a large

Goal

```
← Demo(  
  [ If(Atom(P(0), [V(0), Term(F(0), [V(0), V(1)])]), True),  
    If(Atom(P(0), [V(0), Term(F(0), [V(1), V(2)])]),  
      Atom(P(0), [V(0), V(2)]) ) ]  
  Atom(P(0), [  
    V(0),  
    Term(F(0),  
      [ Term(C(1), []),  
        Term(F(0), [Term(C(2), []), Term(C(0), [])]) ] )  
    ]),  
  g,
```

Computed answers

```
g = Atom(P(0), [  
  Term(C(1), []),  
  Term(F(0),  
    [ Term(C(1), []),  
      Term(F(0), [Term(C(2), []), Term(C(0), [])]) ] ) ]),
```

```
g = Atom(P(0), [  
  Term(C(2), []),  
  Term(F(0),  
    [ Term(C(1), []),  
      Term(F(0), [Term(C(2), []), Term(C(0), [])]) ] ) ])
```

Fig. 7. Goal and computed answers for programs I and J

expression. Hence constructing, changing, or even just querying such an expression can require a large amount of program code. The Gödel system provides considerable system support for the constructing and querying of terms representing object Gödel expressions. By employing an abstract data type approach so that the representation is not made explicit, Gödel allows a user to ignore the details of the representation. Such an abstract data type approach makes the design and maintenance of the meta-programming facilities easier. Abstract data types are facilitated in Gödel by its type and module systems. Thus, in order to describe the meta-programming facilities of Gödel, a brief account of these systems is given.

Each constant, function, predicate, and proposition in a Gödel program

must be specified by a language declaration. The type of a variable is not declared but inferred from its context within a particular program statement. To illustrate the type system, we give the language declarations that would be required for the program in Figure 1.

```

BASE      Name .
CONSTANT Tom, Jerry : Name .
PREDICATE Chase      : Name * Name ;
           Cat, Mouse  : Name .

```

Note that the declaration beginning **BASE** indicates that **Name** is a base type. In the statement

```
Chase(x,y) <- Cat(x) & Mouse(y).
```

the variables **x** and **y** are inferred to be of type **Name**.

Polymorphic types can also be defined in Gödel. They are constructed from the base types, type variables called parameters, and type constructors. Each constructor has an arity ≥ 1 attached to it. As an example, we give the language declarations for the non-logical symbols used in the (second variant) of the program in Figure 2.

```

CONSTRUCTOR List/1.
CONSTANT   Nil    : List(a).
FUNCTION   Cons   : a * List(a) -> List(a)
PREDICATE  Member : a * List(a).

```

Here **List** is declared to be a type constructor of arity 1. The type **List(a)** is a polymorphic type that can be used generically. Gödel provides the usual syntax for lists so that **[]** denotes **Nil** and **[x|y]** denotes **Cons(x,y)**. Thus, if **1** and **2** have type **Integer**, **[1,2]** is a term of type **List(Integer)**.

The Gödel module system is based on traditional software engineering ideas. A program is a collection of modules. Each module has at most two parts, an export part and a local part. The part and name of the module is indicated by the first line of the module. The statements can occur only in the local part. Symbols that are declared or imported into the export part of the module are available for use in both parts of the module and other modules that import it. Symbols that are declared or imported into the local part (but not into the export part) of the module can only be used in the local part. There are a number of module conditions that prevent accidental interference between different modules and facilitate the definition of an abstract data type.

An example of an abstract data type is illustrated in the Gödel program in Figure 8 which consists of two modules, **UseADT** and **ADT**. In **UseADT**, the type **K**, which is imported from **ADT**, is an abstract data type. If the query

```
<- Q(x,D) & P(x,y,z).
```

was given to **UseADT**, then the displayed answer would be:

```

MODULE UseADT.
IMPORT ADT.

EXPORT ADT.
BASE H,K.
CONSTANT C,D : H.
PREDICATE P : H * K * K;
          Q : H * H.

LOCAL ADT.
CONSTANT E : K.
FUNCTION F : H * K -> K.
P(u,F(u,E),E).
Q(C,D).

```

Fig. 8. Defining an abstract data type in Gödel

```

x = C,
y = <K>,
z = <K>

```

The system modules include general purpose modules such as **Integers**, **Lists**, and **Strings** as well as the modules that give explicit support for meta-programming. **Integers** provides the integers and the usual arithmetic operations on the integers. **Lists** provides the standard list notation explained earlier in this subsection as well as the usual list processing predicates such as **Member** and **Append**. The module **Strings** makes available the standard double quote notation for sequences of (ascii) characters. There is no type for an individual ascii character except as a string of length one.

Gödel provides an abstract data type **Unit** defined in the module **Units**. A **Unit** is intended to represent a term-like data structure. The module **Flocks** imports the module **Units** and provides an abstract data type **Flock** which is an ordered collection of terms of type **Unit**. Since **Flocks** and **Units** do not provide any reflective predicates, they cannot be regarded as complete meta-programming modules. However, **Flocks** are useful tools for the manipulation of any object language whose syntax can be viewed as a sequence of units. Thus **Flocks** can be used as the basis of a meta-program that can choose any semantics for the object programming system.

The four system modules for meta-programming are **Syntax**, **Programs**, **Scripts**, and **Theories**. The modules **Programs**, **Scripts**, and **Theories** support the ground representation of Gödel programs, scripts, and theories, respectively. A script is a special form of a program where the module structure is collapsed. Since program transformations frequently violate the module structure, **Scripts** is mainly intended for meta-programs that

perform program transformations. A theory is assumed to be defined using an extension of the Gödel syntax to allow for arbitrary first order formulas as the axioms of the theory. The fourth meta-programming module **Syntax** is imported by **Programs**, **Scripts**, and **Theories** and facilitates the manipulation of expressions in the object language. We describe briefly here the modules **Syntax** and **Programs**. The modules **Scripts** and **Theories** are similar to **Programs** and details of all the meta-programming modules can be found in (Hill & Lloyd 1994).

The module **Syntax** defines abstract data types including **Name**, **Type**, **Term**, **Formula**, **TypeSubst**, **TermSubst**, and **VarTyping** which are the types of the representations of, respectively, the name of a symbol, a type, a term, a formula, a type substitution, a term substitution, and a variable typing. (A *variable typing* is a set of bindings where each binding consists of a variable together with the type assigned to that variable.)

The module **Syntax** provides a large number of predicates that support these abstract data types. Many of these are concerned with the representation and can be used to identify and construct representations of object expressions. For example, **And** is a predicate with three arguments of type **Formula** and is true if the third argument is a representation of the conjunction of the formulas represented in the first two arguments. **Variable** has a single argument of type **Term** and is true if its argument is the representation of a variable.

The predicate **Derive** is an example of a reflective predicate in **Syntax**. **Derive** has the declaration

```
PREDICATE Derive : Formula * Formula * Formula * Formula *
                Formula * TermSubst * Formula.
```

Given atom of the form **Derive**($f_1, f_2, f_3, f_4, f_5, f_6, f_7$), then this is true if r_1 is the resultant derived from a resultant r with selected atom a and statement s using mgu t . Here, r_1 is the resultant represented by f_7 , r is the resultant whose head is represented by f_1 and whose body is a conjunction of the formulas represented by f_2, f_3 , and f_4 ; a is the atom represented by f_3 ; s (whose variables are standardized apart from the variables in r) is the statement represented by f_5 .

The module **Programs** imports the module **Syntax** and defines the abstract data type **Program**. **Program** is the type of a term representing a Gödel program.

As in the module **Syntax**, many of the predicates in **Programs** are concerned with the representation. Some actually relate the object language syntax with the representation. For example, there is a predicate **StringToProgramType/4** with declaration

```
PREDICATE StringToProgramType : Program * String * String *
                               List(Type).
```

that converts a type (represented as a string) to a list of representations of

this type. There may be more than one type corresponding to the string due to overloading of the names of the symbols in the object program. There are other predicates for manipulating the elements of the abstract data type **Program** directly. Thus **DeleteStatement** which has the declaration

```
PREDICATE DeleteStatement : Program * String * Formula *
                          Program.
```

removes a statement from an object program.

Finally, there are several reflective predicates. For example, **Succeed** which has the declaration

```
PREDICATE Succeed : Program * Formula * TermSubst.
```

is true when its first argument is the representation of an object program, its second argument is the representation of the body of a goal in the language of this program, and its third argument is the representation of a computed answer for this goal and this program. The predicate **Succeed** is the Gödel equivalent of the *Demo* predicate.

The program in Figure 9 defines a predicate **SLDDemo/3** using some of the system predicates of **Syntax** and **Programs**. The only reflective system predicate used in this program is **Derive** from **Syntax**. This meta-interpreter works for definite programs and goals and provides a starting point for various extensions.

The SLD-Demo meta-interpreter implements the usual left to right selection rule. However, by changing the above definition of **MyAnd/3**, SLD resolution can be defined with arbitrary selection rules.

Although the predicates in the system modules **Syntax** and **Programs** can be used to construct the representation of a Gödel expression, this method would not be practical for a complete Gödel program. For this reason, there is a utility in the Gödel system that constructs the ground representation of an object program (of type **Program**) and writes this to a file. There is another utility that obtains the re-representation of a term of type **Program** and writes it to the appropriate files.

To read from or write to a file containing a program representation, there is a system module **ProgramsIO** which provides the appropriate input and output system predicates. For example, to run the SLD-Demo program in Figure 9, we can use the module in Figure 10. This assumes that a file containing the ground representation of a program exists. Given a file **Chase.prm** containing a representation of a Gödel version of the Chase program in Figure 1, a query of the form

```
<- Go("Chase", "Chase(Tom, x)",y).
```

has the answer

```
y = "Chase(Tom, Jerry)".
```

```

EXPORT SLDDemo.

IMPORT Programs.
PREDICATE SLDDemo : Program * Formula * Formula.

LOCAL SLDDemo.

SLDDemo(prog, query, query1) <-
  IsImpliedBy(query, query, res) &
  EmptyFormula(empty) &
  SLDDemo1(prog, res, res1) &
  IsImpliedBy(query1, empty, res1).

PREDICATE SLDDemo1 : Program * Formula * Formula.
DELAY SLDDemo1(p,r,_) UNTIL NONVAR(p) & NONVAR(r).
SLDDemo1(_, res, res).
SLDDemo1(prog, res, res1) <-
  EmptyFormula(empty) &
  IsImpliedBy(head, body, res) &
  MyAnd(atom, rest, body) &
  StatementMatchAtom(prog, _, atom, clause) &
  Derive(head, empty, atom, rest, clause, _, newres) &
  SLDDemo1(prog, newres, res1).

PREDICATE MyAnd : Formula * Formula * Formula.
MyAnd(atom, rest, body) <-
  And(atom, rest, body) &
  Atom(atom).
MyAnd(body, empty, body) <-
  EmptyFormula(empty) &
  Atom(body).

```

Fig. 9. An SLD-Demo program using the Gödel meta-programming modules

4 Self-Applicability

There are a number of meta-programs that can be applied to (copies of) themselves. In this section we review the motivation for this form of meta-programming and discuss the various degrees of self-applicability that can be achieved by these programs.

The usefulness of self-applicability was demonstrated by Gödel in (1931) where the natural numbers represent the axioms of arithmetic and some

```

MODULE TestSLDDemo.

IMPORT SLDDemo, ProgramsIO.

PREDICATE Go: String * String * String.
Go(prog_string, goal_string, goal1_string) <-
  FindInput(prog_string ++ ".prm", In(stream)) &
  GetProgram(stream, prog) &
  MainModuleInProgram(prog, module) &
  StringToProgramFormula(prog, module, goal_string, [goal]) &
  SLDDemo(prog, goal, goal1) &
  ProgramFormulaToString(prog, module, goal1, goal1_string).

```

Fig. 10. Gödel program for testing the SLD-Demo program

fundamental theorems about logic are derived using the properties of arithmetic. Perlis and Subrahmanian (1994) give a description of many of the general issues surrounding self-reference in logic and artificial intelligence together with a comprehensive bibliography. The concept of self-applicability has been used to construct many well known logical paradoxes. One of the most famous being the *liar paradox*:

This sentence is false.

In this section we are concerned with programming applications of self-applicability and how particular programming languages support such applications.

If the languages of the object program and meta-program are kept *separate*, we can prevent the problems of self-reference illustrated by the liar paradox above while being able to express the syntactic form of self-reference given in

This sentence has five words.

by replacing the words “This sentence” by a representation of the sentence. Provided a representation of the meta-program can be included as a term in a goal for the meta-program and object provability is defined in the meta-program, the meta-program is self-applicable although the object language and the language of meta-program are kept strictly separate.

Many goals for the meta-program that can also be expressed as goals for the object language can be solved more efficiently using the object program rather than using their representation in the meta-program. In order that the meta-program can use the object program directly, the language \mathcal{M} of the meta-program needs to include the object language \mathcal{L} . Assuming $\mathcal{L} \subseteq \mathcal{M}$, Bowen and Kowalski (1982) defined the *amalgamation* of \mathcal{L} and

\mathcal{M} to be the language \mathcal{M} together with

- at least one ground term in \mathcal{M} representing each term and formula of \mathcal{L} ,
- a representation of the provability relation in \mathcal{L} by means of a predicate in \mathcal{M} (in the context of a set of sentences of \mathcal{M}), and
- linking rules for communicating goals and computed answers between the object language and its representation.

An amalgamation is said to be *strong* if the languages \mathcal{L} and \mathcal{M} are the same. Bowen and Kowalski (1982) showed that self-referential sentences were possible in this logic. They constructed a sentence J that asserts of itself that it is undervivable. They show that neither J nor $\neg J$ is derivable in the logic so that J is clearly true. If \mathcal{L} is a proper subset of \mathcal{M} , then the amalgamation is said to be *weak*. As we are not aware of any logic programming systems that allow the object language and language of the meta-program to be completely identified, only the weak form of amalgamation is discussed in this chapter.

4.1 Separated Meta-Programming

Normally, a self-applicable meta-program reasons about a copy of itself. This can be achieved using both the non-ground and ground representations, described in Sections 2 and 3. By using the meta-program as the object program for another meta-program, meta-meta-programs and higher towers of meta-programs can be constructed.

For example, the Instance-Demo program **I** given in Figure 5 can be applied to the representation of any normal logic program. Thus, as **I** is itself a normal program, **I** can be applied to a representation of itself. This is achieved by giving **I** a goal of the form:

$$\leftarrow IDemo([\mathbf{I}], [goal], g)$$

where *goal* is the goal in Figure 7. We do not give the actual representation of **I** or *[goal]* here, since these terms are large. This difficulty illustrates two of the problems of using towers of meta-programs. The terms used to represent the object program and goal are large and complex and also, because of the added structural information encoded in the terms, processing them is much less efficient than that using the original object code. This issue is discussed more fully in Section 6.

The Gödel system assumes that self-applicable meta-programs are separated. Utilities and certain IO predicates are provided for obtaining the representation and re-representation of a Gödel program. However, apart from these non-declarative facilities, there is no direct access to the object program from a meta-program. In contrast, in Prolog, the object program must be present in the meta-program in order to obtain its representation.

Self-application using separated meta-programming has many applica-

tions and is, we believe, the most useful form of self-applicability. In particular; interpreters, compilers, program analysers, program transformers, program debuggers, and program specializers can be usefully applied to themselves. In Section 3, we explained how the programming system Gödel provided support for meta-programming using the ground representation. This system has been designed so that self-applicable meta-programs can be developed. Towers of two or even three levels of program specializers have been achieved using the Gödel system although much work needs to be done to improve their efficiency (Gurr 1993). Section 6 explains how compilers and even compiler generators can be generated automatically from program specializers using self-application.

4.2 Amalgamated Meta-Programming

It is often desirable that not only is the object language part of the meta-language, but also that the object program is included in the meta-program. Hence we extend the above definition of amalgamation of languages to programs. We say that a meta-program is *amalgamated* if the languages of the meta-program and its object program are amalgamated and the statements of the object program are included as part of the meta-program. With an amalgamated meta-program, a query for the meta-program that represents a query in the object language can be defined directly using the object program. For example, if P is a proposition in the object language represented by the constant P' in the language of the meta-program, then we could have a statement in the meta-program of the form:

$Demo(ThisProgram, P') \leftarrow P.$

In this example and in the examples below, *ThisProgram* is a constant in the meta-language that refers to the (ground) representation of an object program which is included in the meta-program.

A basic requirement for amalgamated meta-programming is that the semantics of the object program must be preserved. That is, the predicates defined in the object program must have the same definition in the amalgamated meta-program so that a goal written in the object language must be a logical consequence of the object program if and only if it is a logical consequence of the amalgamated meta-program.

In order to realise the advantages of amalgamated programming, there has to be a means by which the object program and its representation can communicate. Bowen and Kowalski define the following *linking rules* that should be satisfied by the reflective predicate *Demo/2* in the amalgamated program for every formula B in the object program.

$$\frac{A \vdash B}{Pr \vdash Demo([A], [B])} \qquad \frac{Pr \vdash Demo([A], [B])}{A \vdash B}$$

These or similar rules are necessary for communication between the object program and its representation in the amalgamated meta-program.

To facilitate these linking rules, a means of computing the re-representation of the terms representing the terms of the object language must be provided. Also, a method for finding the representation of an object term is also required. This *reflective requirement*, which concerns only the terms of the object language, may be realised by means of inference rules, functions, or relations. In each case, we consider how the predicate *Demo/2* whose semantics is given by the above linking rules may be defined for the atomic formulas. The definition of *Demo/2* for the non-atomic formulas is the same in every case. Thus, for formulas that are conjunctions of literals, the definition of *Demo/2* would include the following clauses.

$$\begin{aligned} Demo(p, a \wedge' b) &\leftarrow Demo(p, a) \wedge Demo(p, b) \\ Demo(p, \neg' a) &\leftarrow \neg Demo(p, a) \end{aligned}$$

where \wedge' represents \wedge and \neg' represents \neg .

If the reflective requirement is realised by means of inference rules, then these must be built into the programming system. Thus the representation must also be fixed by the programming system. The inference rule that determines t from a term $[t]$ must first check that $[t]$ is ground and that it represents an object level term and secondly, if the object language is typed, that t is correctly typed in this language. The set of statements of the form

$$Demo(ThisProgram, P'([x_1], \dots, [x_n])) \leftarrow P(x_1, \dots, x_n)$$

for each predicate P/n in the object language represented by a function P'/n in the language of the meta-program will provide a definition of *Demo/2* for the atomic formulas. Note that the x_i are universally quantified variables, quantified over the terms of the object language.

Note that, as the trivial naming relation is built into Prolog, the representation is trivially determined by means of an inference rule. However, there is no check that $[t]$ is ground before applying the inference rule. Reflective Prolog (Costantini & Lanzarone 1989) (see below), is an example of a language with a non-trivial naming relation, but where the representation and re-representation are determined by inference rules.

If a re-representation was defined functionally, the meta-program would require a function such as *ReRepresent/1*. Thus, for each ground term t in the object language, the equality theory for the meta-program must satisfy

$$ReRepresent([t]) = t$$

As this is part of a logic programming system where the equality theory is normally fixed by the unification procedure and constraint handling mechanisms, the evaluation method for this function would be built-in so that the representation would again be fixed by the programming system. Using this function, the definition of *Demo/2* for the atomic formulas is given by

a set of statements of the form

$$\begin{aligned} Demo(ThisProgram, P'(y_1, \dots, y_n)) \leftarrow \\ P(ReRepresent(y_1), \dots, ReRepresent(y_n)) \end{aligned}$$

for each predicate P/n in the object program.

For logic programming, the most flexible way in which a representation may be defined is as a relation, say $Represent/2$ where the first argument is an object term and the second its representation.

$$Represent(t, [t]).$$

Then the definition of $Demo/2$, for each predicate P/n in the object language, would consist of a statement of the form

$$\begin{aligned} Demo(ThisProgram, P'(y_1 \dots, y_n)) \leftarrow \\ Represent(x_1, y_1) \wedge \dots \wedge Represent(x_n, y_n) \wedge \\ P(x_1, \dots, x_n) \end{aligned}$$

The predicate $Represent/2$ could be defined by the user and hence, as discussed in Subsection 3.1, chosen to suit a particular application. Thus, for the Member program in Figure 2, we would have the statement

$$\begin{aligned} Demo(ThisProgram, Member'(y_1, \dots, y_n)) \leftarrow \\ Represent(x_1, y_1) \wedge \dots \wedge Represent(x_n, y_n) \wedge \\ Member(x_1, \dots, x_n) \end{aligned}$$

For this example, the definition of $Represent/2$ would include the clauses

$$\begin{aligned} Represent(Nil', Nil) \\ Represent(Cons'(x1, y1), Cons(x, y)) \leftarrow \\ Represent(x1, x) \wedge \\ Represent(y1, y) \end{aligned}$$

In Sections 2 and 3, we gave meta-programs that defined reflective predicates entirely at the meta-level. We have now shown that, if the meta-program is amalgamated, these reflective predicates can be defined using the object program directly. Usually, a combination of these two methods is preferred since the approach that executes a re-representation is usually more efficient, but the explicit representation of the procedural semantics provides greater flexibility. The level at which the explicit definition of the procedural semantics of the object programming system is replaced by a call to the object program using the re-representation determines the granularity of the interpreter. The greater the detail at which the procedure is defined explicitly, the greater the degree of granularity. Clearly efficiency will be decreased with increasing granularity.

Amalgamation not only facilitates greater computational efficiency for meta-programs but also provides an environment that allows interaction between between the actual knowledge and the methods for reasoning about this knowledge. In particular, with an amalgamated language, not only can predicates at the meta-level be defined using object-level predicates,

but also object-level predicates can use meta-level predicates in their definitions. A classic example of an application of this (taken from (Bowen & Kowalski 1982)) is the coding of the legal rule that a person is innocent unless he or she is proven guilty.

$$\begin{aligned} \text{Innocent}(x) \leftarrow \neg \text{Demo}(\text{ThisProgram}, \text{Guilty}'(y)) \wedge \\ \text{Represent}(x, y). \end{aligned}$$

An amalgamation can facilitate towers of meta-programming. For example, a meta-program can include the statement

$$\begin{aligned} \text{Demo}(\text{ThisProgram}, \text{Demo}'(\text{ThisProgram}, y)) \leftarrow \\ \text{Represent}(x, y) \wedge \\ \text{Demo}(\text{ThisProgram}, x). \end{aligned}$$

where the predicate $\text{Demo}/2$ is represented by the function $\text{Demo}'/2$.

Note that at each meta-level, the definition of the predicate $\text{Represent}/2$ must be extended with the constants and functions of the previous meta-level. Suppose, for example, each meta-level uses an additional quote to represent the previous lower level. Then, just to represent the representation of the Member program, the following clauses would need to be added to the above definition of $\text{Represent}/2$.

$$\begin{aligned} \text{Represent}(\text{Nil}'', \text{Nil}') \\ \text{Represent}(\text{Cons}''(x1, y1), \text{Cons}'(x, y)) \leftarrow \\ \text{Represent}(x1, x) \wedge \\ \text{Represent}(y1, y) \\ \text{Represent}(\text{Member}''(x1, y1), \text{Member}'(x, y)) \leftarrow \\ \text{Represent}(x1, x) \wedge \\ \text{Represent}(y1, y) \end{aligned}$$

At the next (third) meta-level, not only would the representation of Nil'' , $\text{Cons}''/2$, and $\text{Member}''/2$ have to be defined by $\text{Represent}/2$ but also the representation of the function $\text{Demo}'/2$.

Each meta-level in a program could contain the representations of several programs at the previous level. The relationships between the different meta-levels in a program are sometimes called its *meta-level architecture*. A meta-program in which each meta-level reasons about only one program at the previous level might be said to have a “linear” architecture.

As the representation has to be explicitly defined using $\text{Demo}/2$ and $\text{Represent}/2$, there is always a top-most meta-level. This will contain symbols with no representation. Hence, without any ‘higher-order’ extensions, logic programming cannot be used for the strong form of amalgamation.

One of the problems of this amalgamation is that the representation needs to be made explicit. In the previous discussion, a representation of the symbols is given and then a representation of the terms and formulas

is constructed in the standard way. However, it is often more convenient to hide the details of the representation from the programmer. Quine (1951) introduced the ‘quasi-quotes’ already used in this chapter to indicate a representation of some unspecified object expression. This (or similar syntax) can be used instead of an explicit representation. For example,

$[Cons(x, Nil)]$

would correspond (using the above representation) to the term

$Cons'(V(0), Nil')$

where $V(0)$ is the (ground) representation of x .

This syntax is not constructive. That is, there is no direct means of constructing larger expressions from their components. Note that, in the Gödel system, the use of abstract data types for meta-programming has a similar problem.

The main reason that terms representing object-level expressions have to be constructed dynamically is because the structure of components of these expressions may not be fully specified. The unspecified sub-expressions are defined by variables that range not over the object terms but over the representations of arbitrary object expressions. Such variables are called *meta-variables*. Thus, instead of a programming system providing predicates for constructing terms representing object expressions, the syntax may distinguish between meta-variables and variables ranging over the object level terms. The partially specified object expression can then be enclosed in the quasi-quotes but those meta-variables that occur within their scope must be syntactically identifiable using some *escape* notation. For example, if the *escape* notation is an overline, \bar{x} indicates that in the context of $[\dots]$ x is a meta-variable. For example,

$[Cons(\bar{x}, Nil)]$

would correspond (using the above representation) to the term

$Cons'(x, Nil')$

where the x ranges over the terms in the language of the Member program. Moreover, $[\bar{x}]$ is equivalent to the meta-variable x^5 . With this notation, the clause in the definition of *Demo/2* that defines the representation of *Member/2* is as follows.

$Demo(ThisProgram, [Member(\bar{x}_1, \dots, \bar{x}_n)]) \leftarrow$
 $Member(x_1, \dots, x_n).$

⁵This differs from Quine's notation where he uses x, y, z etc., to indicate the object variables (in his case, quantified over numbers) and Greek letters to indicate the meta-variables. Using this notation, $[\mu] = \mu$. The disadvantage of this notation is that only two levels are defined and these are assumed to be separated. There is no provision for more than two levels or for amalgamating the meta-levels.

As explained in the previous subsection, Gödel is not intended for amalgamated meta-programming and provides no support for this. Prolog meta-programming facilities force the object program and a meta-program to be amalgamated but the actual switching between the object level and meta-level has to be programmed explicitly. Moreover, most of the meta-programming facilities of Prolog are not declarative whereas the intention of using amalgamated meta-programs for representing knowledge is to provide a declarative representation of this knowledge.

A programming system called *Reflective Prolog* (Costantini & Lanzarone 1989), (Costantini 1990) is intended for amalgamated reasoning. In this language, the representation of the constants, functions, and predicates is defined by specially annotating the corresponding object symbols. Moreover, there are three different kinds of variables: object variables, predicate meta-variables, and function meta-variables. The rules of substitution ensure that these may only be substituted by, respectively, an object term, a representation of a predicate, and a representation of a function. There are syntactic restrictions to keep the meta-levels distinct and prevent self-reference within a single atom. In addition to the object level and different meta-levels, a reflective Prolog program distinguishes between the meta-evaluation level and the base level. The meta-evaluation level is at the top of the meta-level architecture and includes a distinguished predicate `Solve`. The base level, containing an amalgamated theory, comprises the remaining meta-levels below it and cannot refer to any predicates in the meta-evaluation level. Procedurally, a definite Reflective Prolog program uses SLD-resolution whenever possible but automatically switches between the base level and meta-evaluation level in certain circumstances. The declarative semantics for such programs, called the *Least Reflective Herbrand Model*, is an adapted form of the well-known least Herbrand model.

A new system for amalgamated meta-programming called Alloy is under development (Barklund, Boberg & Dell'Aqua 1994). This system is similar to and largely inspired by the ideas in Reflective Prolog. Alloy differs from Reflective Prolog in that it provides explicit support for the ground representation and for facilitating the definition of multiple meta-levels. Although the syntax is not finalised, Alloy intends to employ a syntax based on the quasi-quote and escape notation described above.

Towers of meta-programs and more general meta-level architectures have been shown to be useful in a number of areas. These include software engineering and legal reasoning. Software engineering defines methodologies for program development independent of any particular programming language or application. Tools for supporting these methodologies may distinguish three levels of reasoning. The object level is the application domain. Given the pre-conditions and post-conditions, the first meta-level defines what the program is intended to compute. Finally, the top-most

meta-level defines a formalisation of correct program development. More details concerning this application are given in (Dunin-Keplicz 1994). It is known that legal knowledge has a number of reasoning layers. For example, there may be a several primary legal rules that are intended for distinct situations. Then secondary rules specify when the primary rules are applicable, how to interpret them, or even how to construct new primary rules. In law, such techniques are often applied repeatedly, so that tertiary rules can be defined as meta-rules for the secondary rules and similarly for higher rules. It is shown in (Barklund & Hamfelt 1994) that these layers can be put in a one-to-one correspondence with the meta-levels of an amalgamated meta-program.

4.3 Ambivalent Logic

The flexibility of the trivial naming relation with the non-ground representation in Prolog has encouraged a certain style of meta-programming to be adopted by Prolog programmers. However, first order logic does not provide many Prolog meta-programs with a declarative semantics. For example, a useful feature of Prolog allows an ambivalent syntax where terms and atoms are not distinguished except by their context in the clause, while substitution for variables is purely syntactic. Thus, expressions such as

```
demo(demo(X)) :- demo(X)
demo(X) :- X
```

are allowed. The first of these is easily explained as overloading the name `demo` as both a function and predicate symbol. The second of these can be understood as a schema for clauses of the form

$$\text{demo}(t) \leftarrow t$$

where the argument t is a ground term representing the ground formula t on the right of the arrow. However, the concept of a schema takes the semantics beyond that of first order logic.

There have been a number of proposals for extending first order logic with limited higher order features that may provide these aspects of Prolog with a semantics. Chen, Kifer and Warren (1993) have developed the logic Hilog. This is intended to give a logical basis for Prolog's ambivalent syntax. However, although Hilog does not distinguish between functions, predicates, terms, and atoms, it does not (as in Prolog) allow variables to range over arbitrary expressions in the language. Hilog is intended to provide a basis for a new logic programming system similar to Prolog but based on the logic of Hilog. Richards has defined an ambivalent logic that allows formulas to be treated as terms but not vice-versa (1974). Another *ambivalent logic* is being developed by Jiang (1994). This employs features from both Richards' logic and Hilog. In Jiang's logic, there is no syntactic distinction between functions, predicates, terms, and formulas. Moreover, the semantics distinguishes between substitution (which is purely syntactic)

and equality. The main purpose of this logic is to provide an expressive syntax for self-reference together with a suitable extension of first order logic for its semantics. However, it has also been used to show that the Vanilla program \mathbf{V}_P in Section 2 (without the third clause that interprets negative formulas) has the intended semantics.

5 Dynamic Meta-Programming

We consider three forms of dynamic meta-programming: constructing a program using predefined components, updating a program, and transforming or synthesizing a program.

5.1 Constructing Programs

The simplest and least dynamic means of creating an object program is by combining program components called *modules* to form the object program. There are several applications which require such a modular approach to programming. These include the re-use of existing software, the development of programs incrementally, non-monotonic reasoning, and object-orientation with inheritance.

A number of operators that may be used to construct a complete program from a set of modules are defined in (Brogi, Mancarella, Pedreschi & Turini 1990). These form a sort of “command shell” for building new programs out of existing ones. In (Brogi, Mancarella, Pedreschi & Turini 1992), it is shown how meta-programming techniques using the non-ground representation, can be used to define and implement these operators. We discuss here the use of meta-programming for two of the operators, union \cup and intersection \cap . To explain these, we assume that there is a type *Module* for terms representing sets of clauses, a set of constants of type *Module* representing the sets of clauses that form the initial program components for constructing complete programs, and binary infix operators \cap and \cup with type $\text{Module} * \text{Module} \rightarrow \text{Module}$. Let P_1 and P_2 be two terms of type *Module*. Then:

$P_1 \cup P_2$ represents the module obtained by putting the clauses of modules P_1 and P_2 together.

$P_1 \cap P_2$ represents the module consisting of all clauses defined in the following way:

If $p(t_1, \dots, t_n) \leftarrow B_1$ is in the module represented by P_1 ,
 $p(u_1, \dots, u_n) \leftarrow B_2$ is in the module represented by P_2 ,
and θ is an mgu for $\{p(t_1, \dots, t_n), p(u_1, \dots, u_n)\}$,
then $p(t_1, \dots, t_n)\theta \leftarrow B_1\theta, B_2\theta$ is in the module represented by $P_1 \cap P_2$ ⁶.

⁶We assume that the statements are standardized apart so that they have no variables in common.

. In the non-ground representation described in Section 2, the object program is represented in the meta-program by the definition of the predicate *Clause/2*. With this representation, a declarative meta-program cannot modify the (representation of) the object program. However, by adding an extra argument with type *Module* and replacing *Clause/2* by *OClause/3*, we can include in the representation of a clause the name of the module in which it occurs. For example, given the facts:

OClause(*P*, *Cat*(*Tom*), *True*)
OClause(*P*, *Mouse*(*Jerry*), *True*)
OClause(*Q*, *Chase*(*x*, *y*), *Cat*(*x*) *And* *Mouse*(*y*))

then $P \cup Q$ represents the Chase program in Figure 1. Moreover, with the set of facts:

OClause(*P1*, *Cat*(*Tom*), *True*)
OClause(*P1*, *Mouse*(*Jerry*), *True*)
OClause(*P1*, *Chase*(*x*, *y*), *Cat*(*x*))
OClause(*Q1*, *Cat*(*Tom*), *True*)
OClause(*Q1*, *Mouse*(*Jerry*), *True*)
OClause(*Q1*, *Chase*(*x*, *y*), *Mouse*(*y*))

then $P1 \cap Q1$ also represents the Chase program.

The operators \cup and \cap can be defined by extending program **V** in Figure 3 to give the Operator-Vanilla program **O** in Figure 11. As the operators have only been defined here in the case of definite programs, **O** does not have a clause for interpreting negative literals. Given a set of modules *R*, the program \mathbf{O}_R consists of the program **O** together with the set of facts extending the definition of *OClause* and representing the modules in *R*.

The following result was proved in (Brogi et al. 1992).

Theorem 5.1.1. *Let P and Q be object programs. Then, for any ground atom A in the object language,*

- $\mathbf{O}_{P,Q} \vdash OSolve(P \cup Q, A)$ *iff* A *is a logical consequence of* $P \cup Q$
- $\mathbf{O}_{P,Q} \vdash OSolve(P \cap Q, A)$ *iff* A *is a logical consequence of* $P \cap Q$

For example, given the program in Figure 11 together with the above clauses representing the Chase program, the goals

$\leftarrow OSolve(P \cup Q, Chase(x, y))$
 $\leftarrow OSolve(P1 \cap Q1, Chase(x, y))$

would both succeed with computed answer

$x = Tom, y = Jerry$.

Composing programs with operators allows for programs to be only partly specified. The following result was proved in (Brogi et al. 1990).

$$\begin{aligned}
& OSolve(p, True) \\
& OSolve(p, (b \text{ And } c)) \leftarrow \\
& \quad OSolve(p, b) \wedge \\
& \quad OSolve(p, c) \\
& OSolve(p, a) \leftarrow \\
& \quad OClause(p, a, b) \wedge \\
& \quad OSolve(p, b) \\
& \\
& OClause(p \cup q, a, b) \leftarrow \\
& \quad OClause(p, a, b) \\
& OClause(p \cup q, a, b) \leftarrow \\
& \quad OClause(q, a, b) \\
& OClause(p \cap q, a, (b \text{ And } c)) \leftarrow \\
& \quad OClause(p, a, b) \wedge \\
& \quad OClause(q, a, c)
\end{aligned}$$

Fig. 11. The Operator-Vanilla Interpreter O

Theorem 5.1.2. *Let P be a (possibly non-ground) program term and G represent a goal for the program. Assume that the goal $\leftarrow OSolve(P, G)$ succeeds with computed answer θ . Then the goal represented by $G\theta$ can be proved in the program represented by any ground instance of $P\theta$.*

For example, given the program in Figure 11 together with the above clauses representing the Chase program, the goal

$$\leftarrow OSolve(P \cup q, Chase(x, y))$$

would succeed with computed answers

$$\begin{aligned}
q &= Q, x = Tom, y = Jerry; \\
q &= Q \cup v, x = Tom, y = Jerry; \\
&\vdots
\end{aligned}$$

5.2 Updating Programs

A meta-program can modify an object program by inserting and removing statements. This form of meta-programming has many applications such as hypothetical reasoning, knowledge assimilation, and abduction. For example, in hypothetical reasoning, additional axioms are included as temporary hypotheses during a subcomputation. In knowledge assimilation, the program defining the current knowledge base has to be updated by the addition or deletion of clauses. These changes are usually constrained to satisfy certain integrity constraints as well as to cause minimal changes to the original knowledge base. Abductive reasoning is similar to the previous application except, in this case, only new facts may be added to the object program and no clauses may be deleted. Normally, there is an additional

restriction that there is a predefined set of predicates called *abducibles* and the added facts must (partly) define an abducible predicate.

For these applications, it is necessary, if the meta-program is to be declarative, that the object program be represented as a term in the goal to the meta-program. Moreover, as first order logic does not allow quantifiers in terms, a ground representation must be used.

For knowledge assimilation, the changes to the object program are normally global and permanent. Thus, in the case of knowledge assimilation, it is particularly important that no unnecessary changes are made and the new knowledge base remains consistent. In (Kowalski 1993), the problem of adding a statement S to the database D is discussed in detail. Four cases are described.

1. S is a logical consequence of D
2. $D = D_1 \cup D_2$ and D_2 is a logical consequence of $D_1 \cup \{S\}$.
3. S is inconsistent with D .
4. None of the relationships (1)–(3) hold.

It is assumed here that the knowledge base is a normal logic program whose semantics is taken to be its completion. In this case it is not appropriate to include the consistency checks (since all new facts will be inconsistent with the completion). Thus we only consider cases 1 and 2, and a third case when 1 and 2 do not hold. The program in Figure 12 defining the predicate *Assimilate/3* (which is based on the program in (Kowalski 1990)) shows how these requirements may be realised in logic programming using meta-programming techniques. To simplify the example, we have assumed that only facts may be added or removed. The program uses the representation given in Section 3, where the object program is represented by a list of terms representing the statements of the object program. The program also requires a definition of the reflective predicate *Demo* such as that of *IDemo* given in Figure 5 or *JDemo* given in Figure 6. The types of *Assimilate/3* and *Remove/2* are as follows.

Predicate	Type
<i>Assimilate</i>	$List(\mu) * \mu * List(\mu)$
<i>Remove</i>	$\mu * List(\mu) * List(\mu)$

Remove/3 is true if the formula is an element of the list in the second argument and the third argument is obtained by removing this element.

It is natural to require certain integrity constraints to hold when facts are added to or removed from a knowledge base. These constraints are formulas that should be logical consequences of (the completion of) the updated knowledge base. A set of integrity constraints may be represented as a list of terms. Each term representing an integrity constraint from the set. The assimilation with integrity constraint checking is illustrated in Fig-

$$\begin{aligned}
& \text{Assimilate}(kb, s, kb) \leftarrow \\
& \quad \text{Demo}(kb, s, _) \\
& \text{Assimilate}(kb, s, newkb) \leftarrow \\
& \quad \text{Remove}(\text{If}(a, \text{True}), kb, kb1) \wedge \\
& \quad \text{Demo}([\text{If}(s, \text{True})|kb1], a, _) \wedge \\
& \quad \text{Assimilate}(kb1, s, newkb) \\
& \text{Assimilate}(kb, s, [\text{If}(s, \text{True})|kb]) \leftarrow \\
& \quad \neg \text{Demo}(kb, s, _) \wedge \\
& \quad \neg (\exists a \exists kb1 (\text{Remove}(\text{If}(a, \text{True}), kb, kb1) \wedge \\
& \quad \quad \text{Demo}([\text{If}(s, \text{True})|kb1], a, _))) \\
& \\
& \text{Remove}(x, [x|xs], xs) \\
& \text{Remove}(x, [_|ys], zs) \leftarrow \\
& \quad \text{Remove}(x, ys, zs)
\end{aligned}$$

Fig. 12. Assimilating a Ground Fact into a Database

$$\begin{aligned}
& \text{AssimilateWithIC}([ic|ics], kb, s, newkb) \leftarrow \\
& \quad \text{Demo}([\text{If}(s, \text{True})|kb], ic, _) \wedge \\
& \quad \text{AssimilateWithIC}(ics, kb, s, newkb) \\
& \text{AssimilateWithIC}([], kb, s, newkb) \leftarrow \\
& \quad \text{Assimilate}(kb, s, newkb)
\end{aligned}$$

Fig. 13. Checking Integrity Constraints

ure 13. This defines the predicate *AssimilateWithIC/4* which has the type $List(\mu) * List(\mu) * \mu * List(\mu)$. The first argument of *AssimilateWithIC/4* is the representation of the set of integrity constraints. If the knowledge base, consisting of the initial knowledge base (represented by the second argument) together with the fact which is to be assimilated (represented in the third argument), satisfies the integrity constraints (represented in the first argument), then the *Assimilate* predicate, defined in Figure 12, is called to update the knowledge base. The fourth argument of *AssimilateWithIC/4* contains the representation of the updated knowledge base.

As indicated in Section 3, the first logic programming system to provide declarative facilities for updating logic programs based on a ground representation was Meta-Prolog (Bowen & Weinberg 1985). However, it was built as an extension to Prolog so that it also inherited the non-declarative facilities of Prolog. The system Gödel, described in Subsection 3.3, provides considerable support for updating the theory of a program using its ground representation. For example, the system module **Programs** defines predicates **InsertStatement/4** and **DeleteStatement/4** for adding a statement to and removing a statement from a module in a program.

5.3 The Three Wise Men Problem

We illustrate the use of meta-programming for hypothetical reasoning by means of a well-known problem, *the three wise men*. This problem has been much studied by researchers in meta-reasoning and it is thought appropriate to show here how the standard techniques of meta-programming in logic programming can be used to solve this problem.

The three wise men puzzle is as follows. A king, wishing to find out which of his three wise men is the wisest, puts a hat on each of their heads and tells them that each hat is either black or white and at least one of the hats is white. The king does this in such a way that each wise man can see the hats of the other wise men, but not his own. In fact, each wise man has a white hat on. The king then successively asks each wise man if he knows the colour of his own hat. The first wise man answers “I don’t know”, as does the second. Then the third announces that his hat is white.

The reasoning of the third wise man is as follows. “Suppose my hat is black. Then the second wise man would see a black hat and a white hat, and would reason that, if his hat is black, the first wise man would see two black hats and hence would conclude that his hat is white since he knows that at least one of the hats is white. But the second wise man said he didn’t know the colour of his hat. Hence my hat must be white.”

The solution below, using pure logic programming, is intended to illustrate the use of meta-programming for hypothetical reasoning. In this solution, the king uses his knowledge to simulate the reasoning of the three wise men ($W1$, $W2$, $W3$). The king assumes that the third wise man uses his reasoning to simulate the reasoning of the second wise man and hence of the first. We use the ground representation given in Subsection 3.1 and assumed by the programs in Figures 5, 6, and 13. The constants and predicates used for the king’s and wise men’s knowledge bases together with their representation is as follows:

Object Symbol	Representation
<i>Black</i>	$C(0)$
<i>White</i>	$C(1)$
$W1$	$C(11)$
$W2$	$C(12)$
$W3$	$C(13)$
<i>Hat/2</i>	$P(1)$
<i>DontKnow/1</i>	$P(2)$
<i>Hear/2</i>	$P(3)$
<i>See/2</i>	$P(4)$
<i>DiffColor</i>	$P(5)$
<i>Same/2</i>	$P(6)$

This program assumes that definitions of *Demo* such as that of *IDemo* in

Figure 5 or *JDemo* in Figure 6 and of *AssimilateWithIC* as given in Figure 13 are included. The knowledge of the king and his wise men consists of sets of normal clauses represented by a list of representations of these clauses in some order. There are two initial knowledge bases. One contains knowledge that is common to all the men in the story.

```

Hat(W3, White) ← Hat(W2, Black) ∧ Hat(W1, Black)
Hat(W2, White) ← Hat(W1, Black) ∧ Hat(W3, Black)
Hat(W1, White) ← Hat(W3, Black) ∧ Hat(W2, Black)
Hear(W3, W2)
Hear(W3, W1)
Hear(W2, W1)
See(x, y) ← ¬Same(x, y)
DiffColor(White, Black)
DiffColor(Black, White)
Same(x, x)

```

This is represented by the single fact defining the predicate *CommonKb/1*.

```

CommonKb([
  If(Atom(P(1), [C(13), C(1)]),
    And(Atom(P(1), [C(12), C(0)]), Atom(P(1), [C(11), C(0)]))),
  If(Atom(P(1), [C(12), C(1)]),
    And(Atom(P(1), [C(11), C(0)]), Atom(P(1), [C(13), C(0)]))),
  If(Atom(P(1), [C(11), C(1)]),
    And(Atom(P(1), [C(13), C(0)]), Atom(P(1), [C(12), C(0)]))),
  If(Atom(P(3), [C(13), C(12)]), True),
  If(Atom(P(3), [C(13), C(11)]), True),
  If(Atom(P(3), [C(12), C(11)]), True),
  If(Atom(P(4), [V(1), V(2)]), Not(Atom(P(6), [V(1), V(2)]))),
  If(Atom(P(5), [C(1), C(0)]), True),
  If(Atom(P(5), [C(0), C(1)]), True),
  If(Atom(P(6), [V(1), V(1)]), True)
])

```

In addition to the above common knowledge base, the men will have certain commonly held constraints on what combination of knowledge is acceptable. As an example of such a constraint, we assume that all men know that a hat cannot be both black and white.

```

¬(Hat(w, Black) ∧ Hat(w, White))

```

This can be represented in the fact defining the predicate *CommonIC/1*.

```

CommonIC([
  Not(And(
    Atom(P(1), [V(11), C(0)]),
    Atom(P(1), [V(11), C(1)])))
]).

```

The other knowledge base contains a list of facts describing the King's knowledge about the state of the world.

Hat(W1, White)
Hat(W2, White)
Hat(W3, White)
DontKnow(W1)
DontKnow(W2)

This is represented by the fact defining the predicate *KingKb/1*.

KingKb(
 If(*Atom*(*P*(1), [*C*(11), *C*(1)]), *True*),
 If(*Atom*(*P*(1), [*C*(12), *C*(1)]), *True*),
 If(*Atom*(*P*(1), [*C*(13), *C*(1)]), *True*),
 If(*Atom*(*P*(2), [*C*(12)]), *True*),
 If(*Atom*(*P*(2), [*C*(11)]), *True*)
)

The king and the wise men can use only their own knowledge when simulating other men's reasoning. The key to their reasoning is defined by the predicate *LocalKb/3*. Given the name of a wise man and a current knowledge base in the first and second arguments, respectively, then *LocalKb/3* will be true if the third argument is bound to the part of the knowledge base of the wise man which is contained in the current knowledge base.

LocalKb(*w, kb, localkb*) \leftarrow *LocalKb1*(*w, kb, kb, localkb*)

LocalKb1(*_, _, [], []*)
LocalKb1(*w, ikb, [k|kb], [k|lkb]*) \leftarrow
 CommonKb(*ckb*) \wedge
 Member(*k, ckb*) \wedge
 LocalKb1(*w, ikb, kb, lkb*)
LocalKb1(*w, ikb, [If*(*Atom*(*P*(1), [*w1, c*]), *True*)|*kb*,
 [*If*(*Atom*(*P*(1), [*w1, c*]), *True*)|*lkb*]) \leftarrow
 Demo(*ikb, Atom*(*P*(4), [*w, w1*]), $_$) \wedge
 LocalKb1(*w, ikb, kb, lkb*)
LocalKb1(*w, ikb, [If*(*Atom*(*P*(1), [*w1, _*]), *True*)|*kb*,
 lkb) \leftarrow
 \neg *Demo*(*ikb, Atom*(*P*(4), [*w, w1*]), $_$) \wedge
 LocalKb1(*w, ikb, kb, ikb, lkb*)
LocalKb1(*w, ikb, [If*(*Atom*(*P*(2), [*w1*]), *True*)|*kb*,
 [*If*(*Atom*(*P*(2), [*w1*]), *True*)|*lkb*]) \leftarrow
 Demo(*ikb, Atom*(*P*(3), [*w, w1*]), $_$) \wedge
 LocalKb1(*w, ikb, kb, lkb*)
LocalKb1(*w, ikb, [If*(*Atom*(*P*(2), [*w1*]), *True*)|*kb*,
 lkbs) \leftarrow

$$\neg Demo(ikb, Atom(P(3), [w, w1]), _) \wedge \\ LocalKb1(w, ikb, kbs, lkb)$$

The predicate *LocalKb/3* calls *LocalKb1/4*. This predicate requires an extra copy of the initial knowledge base so that it can process all clauses represented by elements of the list and determine which should be included in the new knowledge base. There are six statements defining *LocalKb1/4*. The first is the base case. The second ensures that the common knowledge is included in the wise man's knowledge. The third (resp., fourth) deals with the case where the wise man can (resp., cannot) see a hat and the colour is known. The fifth (resp., sixth) deals with the case where the wise man can (resp., cannot) hear another man and that man says he doesn't know.

The predicate *Reason/3* simulates the reasoning of the men. Either the man in the first argument reasons that colour of his hat is white because he believes the other two hats are black, or, if he has heard another wise man say "I do not know the colour of my hat", he hypothesises a colour for his own hat and by simulating the other man's reasoning, tries to obtain a contradiction. We use the predicate *AssimilateWithIC* defined in Figure 13 to update the knowledge base. This checks that no integrity constraints are violated by the extra hypothesis.

$$Reason(kb, w, c) \leftarrow \\ Demo(kb, Atom(P(1), [w, V(1)]), Atom(P(1), [w, c])) \\ Reason(kb, w, c1) \leftarrow \\ Demo(kb, Atom(P(2), [V(1)]), Atom(P(2), [w1])) \wedge \\ LocalKb(w1, kb, newkb) \wedge \\ Demo(kb, Atom(P(5), [V(1), V(2)]), Atom(P(5), [c1, c2])) \wedge \\ CommonIC(ics) \wedge \\ AssimilateWithIC(ics, newkb, Atom(P(1), [w, c2]), newkb1) \wedge \\ Reason(newkb1, w1, _)$$

Finally, the predicate *King/2* models the king's own reasoning. The king will deduce that the man in the first argument should be able to reason that the colour of his hat is the colour given in the second argument.

$$King(w, c) \leftarrow \\ KingKb(kkb) \wedge \\ CommonKb(ckb) \wedge \\ Append(kkb, ckb, kb) \wedge \\ LocalKb(w, kb, lkb) \wedge \\ Reason(lkb, w, c)$$

With the program consisting of these definitions together with the programs in figures 5, 12, and 13 and the usual definitions of *Append/3* and *Member/2*, the goal

$$\leftarrow King(C(13), c)$$

has just the computed answer

$c = C(1)$.

Moreover, the goals

$\leftarrow King(C(11), c)$

$\leftarrow King(C(12), c)$

both fail⁷.

Note that this program is not amalgamated, even in the weak sense, and only requires one level of meta-programming.

A number of alternative approaches to the solution of this problem have been made. Kim and Kowalski (1990) use full first order logic to give a representation and solution of the problem. This solution requires meta-level reasoning with a weak amalgamation. The solution in (Aiello, Nardi & Schaerf 1988) also uses full first order logic but is designed for a more general framework of non-cooperative but loyal and perfect reasoners. Their solution has been machine-checked using a version of Weyhrauch's FOL system (Weyhrauch 1982). Nait Abdallah (1987) extends logic programming with a concept similar to a module called an *ion*. This solution is not expressed as a logic program and does not use meta-programming techniques. An alternative solution similar to the one given above using the Gödel system is in (Hill & Lloyd 1994). This was primarily designed to illustrate the meta-programming facilities in Gödel and differs in that its object program defining the wise men's knowledge is purely propositional and a query may only ask the third wise man the colour of his hat.

5.4 Transforming and Specializing Programs

A major application for meta-programming is in the transformation and specialization of programs. However, it appears that although many transformation and specialization procedures and their proofs of correctness have been published, apart from the work of Gurr (1993), little work has been done to establish good declarative meta-programming styles for implementing such procedures.

When a program is specialized for a particular application or transformed to a more efficient program, although the new program may have little resemblance to the old program, it should preserve the semantics, at least with respect to the expected goals for the program. It is quite clear that the old object program has to be represented as a ground term in the goal to the meta-program. In addition, the computed answer should bind a variable in the goal to a ground term representing the transformed or specialized program.

⁷The wise men program with these goals has been machine-checked using the Gödel system.

It is difficult to discuss how program transformers and specializers can be implemented in logic programming without a concrete procedure in mind. Thus we consider the meta-programming requirements for a basic unfolding step and provide a skeleton logic program that realises this step. To define an unfolding step, it is convenient to denote the body of a normal clause as a sequence of literals or conjunctions of literals. Thus, for example,

$$H \leftarrow L_1, \dots, L_n$$

denotes the normal clause

$$H \leftarrow L_1 \wedge \dots \wedge L_n.$$

The unfolding step for definite programs is defined as follows (Tamaki & Sato 1984)⁸.

Definition 5.4.1. Let P be a normal program that includes the clause

$$C : \quad A \leftarrow S, Q(\underline{t}), R.$$

Suppose, for $i = 1, \dots, r$, the clauses

$$D_i : \quad Q(\underline{t}_i) \leftarrow R_i$$

are variants (chosen to have no variables in common with C) of all the clauses in P whose heads unify with $Q(\underline{t})$. Let θ_i be the mgu of $Q(\underline{t})$ and $Q(\underline{t}_i)$. Then the *result of unfolding C with respect to $Q(\underline{t})$* is the program P' obtained from P by replacing C by the r clauses

$$D'_i : \quad A\theta_i \leftarrow S\theta_i, R_i\theta_i, R\theta_i$$

obtained by resolving C with D_i wrt $Q(\underline{t})$.

This definition was first proved correct by Tamaki and Sato (1984) for definite programs and later by Seki (1989) and Gardner and Shepherdson (1991) for normal programs. Kanamori and Horiuchi (1987) showed that it preserves computed answers.

Figure 14 contains the top part of a program that performs an unfolding step. The ground representation given in Subsection 2.1 is assumed. The types of *Unfold/4*, *SelectClause/2*, *DeriveAll/5*, and *Replace/4* are as follows.

Predicate	Type
<i>Unfold</i>	$\mu * \mu * List(\mu) * List(\mu)$
<i>SelectClause</i>	$\mu * List(\mu)$
<i>DeriveAll</i>	$\mu * \mu * List(\mu) * List(\mu) * Integer$
<i>Replace</i>	$List(\mu) * \mu * \mu * List(\mu)$

The predicate *SelectClause/2* selects a clause from a program; *DeriveAll/5* attempts a derivation step for every clause in the program; and *Replace/4*

⁸A tuple t_1, \dots, t_j is denoted here by \underline{t} .

$$\begin{aligned}
& \text{Unfold}(\text{If}(h, b), a, p, q) \leftarrow \\
& \quad \text{SelectClause}(\text{If}(h, b), p) \wedge \\
& \quad \text{SelectLit}(\text{Atom}(P(n), xs), b) \wedge \\
& \quad \text{MaxForm}(\text{If}(h, b), n) \wedge \\
& \quad \text{DeriveAll}(\text{If}(h, b), a, p, cs, n) \wedge \\
& \quad \text{Replace}(p, \text{If}(h, b), cs, q) \\
\\
& \text{DeriveAll}(_, _, [], [], _) \\
& \text{DeriveAll}(\text{If}(h, b), a, [pc|pcs], [c|cs], m) \leftarrow \\
& \quad \text{Resolve}(b, a, pc, [], s, b1, m, n) \wedge \\
& \quad \text{ApplyToForm}(s, \text{If}(h, b1), c) \wedge \\
& \quad \text{DeriveAll}(\text{If}(h, b), a, pcs, cs, n) \\
& \text{DeriveAll}(\text{If}(h, b), a, [pc|pcs], cs, m) \leftarrow \\
& \quad \neg \text{Resolve}(b, a, pc, [], _, _, m, _) \wedge \\
& \quad \text{DeriveAll}(\text{If}(h, b), a, pcs, cs, m) \\
\\
& \text{Resolve}(q, \text{Atom}(P(k), xs), \text{If}(\text{Atom}(P(k), ys), ls), s, t, r, m, n) \leftarrow \\
& \quad \text{Rename}(m, \text{If}(\text{Atom}(P(k), ys), ls), \text{If}(\text{Atom}(P(k), y1s), l1s), n) \wedge \\
& \quad \text{UnifyTerms}(y1s, xs, s, t) \wedge \\
& \quad \text{ReplaceConj}(q, \text{Atom}(P(k), xs), l1s, r)
\end{aligned}$$

Fig. 14. The Unfolding Program

removes an element from a list and inserts a sublist of elements in its place. The remaining predicates are described in Subsection 3.2.

The unfold procedure requires certain basic steps: standardising apart the variables used in the program's clauses from the variables in the clause selected for unfolding, computing a unifier, and applying substitutions, and so on. Moreover at the heart of such a procedure is the need to construct a list of the representations of all clauses that satisfy certain conditions. This could also be achieved by means of intensional sets (if these are provided) and then converting the set of clauses to a list. These basic steps are common to program transformers and specializers and many other similar meta-programs. Thus to simplify the writing of this kind of meta-programming application, predicates that perform these tasks should be provided by the meta-programming system.

Frequently, when transforming a program, the language has to be extended with new functions and predicates. The names of these symbols need to be generated by the meta-program. Thus, the meta-programming system needs to provide support for constructing new names for symbols and modifying the language of an object program. The representation given in Subsection 3.1 does not include a representation of the characters in the symbols of the object language and hence the representation is

not adequate for program transformers that extend or change the object language.

The language of a Prolog program is determined by the functions and predicates used in the clauses and goal. Also, names of symbols can be converted to and from their corresponding lists of ascii codes by means of predicates such as `name/2`. Hence, new symbol names can be added to the representation of the object language. As the naming relation is trivial, this defines new symbol names for the object languages themselves. However, as Prolog does not support a ground representation, writing declarative meta-programs for transforming programs is extremely difficult.

The Gödel system not only includes predicates for adding and deleting statements, but also for changing the object language. For example, the system module `Programs` defines the predicate `ProgramConstantName/4` which will convert between a string of characters forming the name of a constant in an object language and its representation. The module `Strings` provides standard string processing predicates and functions. Thus, new names of symbols required for defining new object languages can be created and their representation obtained. Other predicates add and delete representations of names of symbols and their declarations to and from the representation of an object language. For example, the predicate `InsertProgramConstant/6` creates a representation of a new program from the representation $[P]$ of an existing program P by adding the representation of a constant to $[P]$. Moreover, the system module `Syntax` in Gödel provides many predicates for basic meta-programming tasks such as standardising apart, applying a substitution, and finding an mgu.

In Section 6, the use of meta-programming for program specialization is illustrated by partial evaluation. This technique uses partial information about the goal to create a specialized program. It is intended that the specialized program has the same semantics as the original program (when called with an appropriate goal) but improved efficiency.

6 Specialization of Meta-Programs

Meta-level computations involve an overhead for interpreting the representation of an object program. The more complex and expressive the representation, the greater the overhead is likely to be. The ground representation, in particular, is associated by many with inefficiency. In this section we discuss this issue and see that the overhead can be “compiled away” for a meta-program operating on a given object program.

The method for doing this is based on a program transformation technique called program specialization. This is a large topic in its own right, and is not limited to meta-programs in its application. But the combination of meta-programming and program specialization appears a particularly fruitful one, and so we discuss it and its applications.

6.1 Logic Program Specialization

Let P be a logic program and G a goal. The aim of specialization is to derive another program P' say, whose computations with G (and instances of G) give identical results to those given by P . For goals other than G and its instances, P' may give different results. The restriction of P to G is exploited to gain efficiency; in other words, P' should be more efficient than P , with respect to G .

The topic has been studied in a variety of programming languages, and is often called partial evaluation. A comprehensive treatment of partial evaluation, mainly for functional languages but with some sections on other languages, is given by Jones, Gomard, and Sestoft (1993). Partial evaluation was introduced into logic programming by Komorowski (1982) (who called it partial deduction), and the basic principles and results were established by Lloyd and Shepherdson (1991). A recent survey of techniques and results can be found in (Gallagher 1993).

Our main interest in specialization is when P , the program to be specialized, is a meta-program. Suppose either P or the goal G with respect to which P is to be specialized contains the representation of an object program. In this case the aim of the specialization of P with respect to G is to compile away the representation of the object program.

In the following sections we show the specialization of the Instance-Demo interpreter and a resolution procedure applied to fixed object programs. The specialization of the Instance-Demo program yields clauses syntactically isomorphic to the object language clauses, and therefore computations using the specialized *IDemo/3* are almost identical to computations of the object program. The specialization of resolution yields a program containing low-level predicates manipulating terms, substitutions and so on. In fact specialization of a resolution interpreter is close to true compilation of the object program to a lower-level target language, and the low-level operations correspond to instructions in the target language.

6.1.1 Specialization of the Instance-Demo Interpreter

The Instance-Demo program **I** in Figure 5 can be partially evaluated with respect to a given object program. Figure 15 shows again the top level procedures of this interpreter. When supplied with an object program P , **I** can be specialized by partial evaluation, with respect to the goal $\leftarrow IDemo([P], x, y)$. If the object program is the Member program (Figure 2) then the goal is as follows.

```
 $\leftarrow IDemo([$   
     $If(Atom(P(0), [Var(0), Term(F(1), [Var(0), Var(1)]))], True),$   
     $If(Atom(P(0), [Var(0), Term(F(1), [Var(1), Var(2)]))],$   
         $Atom(P(0), [Var(0), Var(2)]))],$   
     $x, y)$ 
```

$$\begin{aligned}
IDemo(p, x, y) &\leftarrow InstanceOf(x, y) \wedge IDemo1(p, y) \\
IDemo1(-, True) &. \\
IDemo1(p, And(x, y)) &\leftarrow \\
&IDemo1(p, x) \wedge \\
&IDemo1(p, y) \\
IDemo1(p, Not(x)) &\leftarrow \\
&\neg IDemo1(p, x) \\
IDemo1(p, Atom(P(n), xs)) &\leftarrow \\
&Member(z, p) \wedge InstanceOf(z, If(Atom(P(n), xs), b)) \wedge \\
&IDemo1(p, b)
\end{aligned}$$

Fig. 15. The Instance-Demo Interpreter I

$$\begin{aligned}
IDemo([& \\
&If(Atom(P(0), [Var(0), Term(F(1), [Var(0), Var(1)])]), True), \\
&If(Atom(P(0), [Var(0), Term(F(1), [Var(1), Var(2)])]), \\
&Atom(P(0), [Var(0), Var(2)])), \\
&x, y) \leftarrow \\
&InstanceOf(x, y) \wedge \\
&IDemo1(y). \\
IDemo1(Atom(P(0), [x, Term(F(1), [x, z])]) &. \\
IDemo1(Atom(P(0), [x, Term(F(1), [y, z])]) &\leftarrow \\
IDemo1(Atom(P(0), [x, z])) &.
\end{aligned}$$

Fig. 16. The Specialized Instance-Demo Interpreter

A suitable partial evaluation of **I** with respect to this goal gives the result shown in Figure 16.

The example shows that substantial optimizations are obtainable by partial evaluation, since the overhead of handling the ground representation in the original program has been almost eliminated. In other words, computations with the specialized Instance-Demo program are almost identical to object-level computations with the object program. In the specialized program the clauses for *IDemo1/2* are very similar to the clauses in the object program representation, except that the representations of variables have been replaced by meta-variables. This arises since the calls to *Member(z, p)* and *InstanceOf(z, If(Atom(P(n), xs), b))* have been completely unfolded, where *p* is the representation of the *Member* program. Secondly, the first argument of *IDemo1/2* containing the object program has been completely eliminated from the specialized *IDemo1/2* predicate

by means of a well-known structure specialization applied by most logic program specialization systems (Gallagher & Bruynooghe 1990). Note that the Instance-Demo program (and hence also its partial evaluation) is typed and this ensures that the new meta-variables range over representations of object terms.

It is interesting to compare the natural interpretations of the original Instance-Demo program and the specialized program in Figure 16. In the original Instance-Demo program the intended domain of interpretation is the set of object language expressions, and the denotation of a term in the ground representation is the object term that it represents. On the other hand in the specialized program, the natural domain of interpretation is (an extension of) the domain of the Member program.

At first sight it is odd that partial evaluation has the effect of changing the intended interpretation of the Instance-Demo program as well as specializing its behaviour. This effect comes about since the general Instance-Demo program handles arbitrary programs in the object language, and the interpretation of the object language is unknown. But specialization with respect to a particular object program allows, indeed suggests, a particular interpretation of the object language, which induces a natural interpretation of the meta-language representations. In particular, if the interpretation of a ground object term t is an object d in the domain of interpretation of the object language, then $[t]$ in the meta-language also denotes d . For terms $[t]$ where t is a non-ground object expression a reasonable interpretation of $[t]$ is the set of objects denoted by ground instances of t . Note that this interpretation gives non-standard, but quite reasonable interpretations for predicates such as *InstanceOf*/2, which are now interpreted as relations on the domain of the Member program rather than on the domain of object language expressions.

It is clear from the example above that a corresponding set of clauses for *IDemo*1/2 could be derived for any object program P . Suppose P is of form $[If(h_1, b_1), \dots, If(h_n, b_n)]$, where h_1, \dots, h_n and b_1, \dots, b_n are the representations of the heads and bodies of the clauses respectively. The specialized Instance-Demo program contains a set of clauses $\{IDemo1(h'_1) \leftarrow IDemo1(b'_1), \dots, IDemo1(h'_n) \leftarrow IDemo1(b'_n)\}$, where each h'_i and b'_i is obtained from h_i and b_i respectively by replacing the ground representation of variables by typed meta-variables.

Kowalski (1990) sketched the derivation of Solve-style interpreter containing non-ground clauses representing an object program, by unfolding a Demo interpreter. Kowalski's interpreter was similar to the Instance-Demo interpreter, but contained a set of ground unit clauses representing the object program instead of representing the program in an argument of *Demo*. The derivation was achieved by unfolding a call to a predicate performing substitution. The use of types (or corresponding sort predicates) in the Instance-Demo program is also essential to the process, a point missed

by Kowalski, since otherwise the meta-variables may have unintended instances. Kowalski's argument, and the partial evaluation example above, suggest the conclusion that the typed non-ground representation of an object program is conceptually not far removed from a ground representation. The limitations of the Solve-style interpreters follow from the limitations of the Instance-Demo program.

However, as a technique for gaining efficiency in the ground representation, the partial evaluation of Instance-Demo is interesting. There is a superficial connection with the *ad hoc* implementation technique of "melting" ground representations by substituting meta-variables in place of ground representations of variables, in order to increase efficiency of meta-programming with the ground representation. This was used for example in the Logimix self-applicable partial evaluator for Prolog (Mogensen & Bondorf 1993)).

6.1.2 Specialization of a Resolution Procedure

More complex and flexible interpreters than Instance-Demo (such as a resolution proof procedure) are more difficult to specialize effectively. Gurr (1993) achieved substantial efficiency improvements by partial evaluation of a resolution interpreter written for the ground representation in Gödel. This work and others (Kursawe 1987), (Nilsson 1993) aimed to show that specialization of logic program interpreters can produce results similar to standard techniques for compiling logic programs based on the Warren Abstract Machine. The connection between specialization and compilation will be further discussed in Section 6.2. Here we illustrate the idea using an example of the partial evaluation of resolution in Gurr's system.

Gurr partially evaluated a predicate *Resolve/7* which we have already used in the SLD-Demo program in Figure 9. It is intended that the first argument of *Resolve/7* represents a formula g ; the second, a program clause c ; the third, a substitution $s1$ to be applied to g ; the fifth, the resolvent $g1$ of g and c (with substitution $s1$ applied to g); and the fourth, the resulting substitution $s2$. The remaining arguments $v1$ and $v2$ are indices for renaming variables during standardization apart. We do not show the code for *Resolve/7* used by Gurr, which incorporates standardization apart, unification of the goal with the clause head, application of the substitution to the clause body, and composition of the unifier with the input substitution.

During partial evaluation of the resolution interpreter with a given object program, the aim is to generate a separate specialized call to *Resolve* for each clause in the object program. Consider the example used in (Gurr 1993) where the object program contains a clause C :

$$P(x, x, A, F(y, F(x, A))) \leftarrow Q(y)$$

Figure 17 shows the call to *Resolve/7* that is to be partially evaluated. The

```

← Resolve(g,
  If(Atom(P(0), [V(0), V(0), Term(C(0), []),
    Term(F(0), [V(1), Term(F(0), [V(0), Term(C(0), [])])))]),
    Atom(P(1), [V(1)])),
  s1, s2, g1, v1, v2)

```

Fig. 17. Instantiated Call to *Resolve*

```

ResolveC(Atom(P(0), [arg1, arg2, arg3, arg4]),
  subst_in, subst_out,
  Atom(P(1), [var]),
  v, v1) ←
  UnifyTerms(arg1, arg2, subst_in, s1) ∧
  GetConstant(arg3, Term(C(0), []), s1, s2) ∧
  GetFunction(arg4, Term(F(0), [t1, t2]), mode, s2, s3) ∧
  UnifyVariable(mode, t1, var, v, v1) ∧
  UnifyFunction(mode, t2, Term(F(0), [t21, t22]), mode1, s3, s4) ∧
  UnifyValue(mode1, arg1, t21, s4, s5) ∧
  UnifyConstant(mode1, t22, Term(C(0), []), s5, subst_out)

```

Fig. 18. Partial Evaluation of *Resolve*

ground representation defined in Section 3 is used, where $P(0)$ represents P , $P(1)$ represents Q , $C(0)$ represents A , $F(0)$ represents F and $V(0), V(1)$ represent x and y respectively. Figure 18 shows the result. A new predicate $Resolve_C/6$ has been created and the representation of the clause C (the second argument of $Resolve$) has been eliminated. The calls in the body to $UnifyTerms$, $GetConstant$, $GetFunction$, $UnifyVariable$, $UnifyConstant$ and $UnifyValue$ are the operations corresponding to the detailed matching of the parts of the head of the clause, and are the residual parts of the full resolution procedure written by Gurr. Note that the variable names from the clause have also completely disappeared since the standardization apart has effectively been carried out during partial evaluation. The body predicates are deliberately named after their analogous Warren Abstract Machine (WAM) instructions. This emphasises the fact that partial evaluation with respect to a given object program achieves a similar result to compilation of the program.

We can compare specialization with the other main approach to improving efficiency in meta-programs, namely “meta-programming facilities”, such as those provided in Gödel or Reflective Prolog. Meta-programming facilities consist of essential meta-programming tools and commonly-used procedures that are carefully coded, and provided to the programmer in

libraries or as built-in procedures in the meta-language. Such facilities are specific to a given object language and a representation of it. This helps to avoid unnecessary inefficiency, but does not eliminate the overhead of using the representation. The advantage of such tools is that once written they are easily applicable. The disadvantage is that there will always be applications outside the scope of the given set of meta-programming facilities. For example, if one has to deal with object programs in a different language, or use a different representation, the built-in facilities may not be applicable.

Specialization has the advantage of being applicable to any object language and representation. Its disadvantage is that it has to be applied to each meta-program separately, though the Futamura projections discussed below alleviate this overhead.

Ideally, both approaches to efficiency improvement are combined. A meta-programming system should provide some facilities for efficient handling of standard meta-programming problems, together with program specialization tools. The latter can gain further efficiency for the standard procedures, and also help to optimize meta-programs outside the scope of the standard facilities. This complementary approach to optimizing meta-programs suggests that built-in procedures should be written in such a way as to make them more “specializable” (Gurr 1993), though what this means in practice is not yet completely understood.

6.2 Specialization and Compilation

The specialization of meta-programs is analogous to compilation, and the relation between executing specialized and unspecialized meta-programs is analogous to the relation between running compiled and interpreted code. The comparison with compilation is actually quite extensive and will be discussed further below. A functional notation is used below though an equivalent, but more cumbersome formulation in logic programming is possible.

Definition 6.2.1. A program specializer *written in \mathcal{M} for \mathcal{L}* is a function

$$PS_{\mathcal{M}} : [\mathcal{L}]_{\mathcal{M}} \rightarrow ([\mathcal{L}]_{\mathcal{M}} \rightarrow [\mathcal{L}]_{\mathcal{M}})$$

We assume that an object program $p_{\mathcal{L}}$ in \mathcal{L} is a function with two arguments, and that the first argument is known but not the second. The specializer takes the representations of $p_{\mathcal{L}}$ and some data $x_{\mathcal{L}}$ for the first argument of $p_{\mathcal{L}}$ and computes the representation of a specialized program $p_{\mathcal{L}}^x$.

The defining property of the specializer PS is the following:

$$PS_{\mathcal{M}}([p], [x]_{\mathcal{M}}) = [p^x]_{\mathcal{M}} \Rightarrow (\forall y) p(x, y) = p^x(y)$$

Definition 6.2.2. A language interpreter *written in \mathcal{M} for \mathcal{L}* is a function

$$I_{\mathcal{M}} : [\mathcal{L}]_{\mathcal{M}} \rightarrow ([D]_{\mathcal{M}} \rightarrow [D]_{\mathcal{M}})$$

(We assume that D is the language of input and output for programs in \mathcal{L}). That is, $I_{\mathcal{M}}$ takes the representations of a program $p_{\mathcal{L}}$ and some data x_D for $p_{\mathcal{L}}$ and computes the representation of the output, say y_D .

The analogy between compilation and interpreter specialization was first identified by Futamura (Futamura 1971). In the following we formulate the so-called Futamura projections in such a way as to emphasize meta-programming aspects.

Definition 6.2.3. First Futamura Projection

Let $PS_{\mathcal{K}}$ be a program specializer written in \mathcal{K} for \mathcal{M} . Let $I_{\mathcal{M}}$ be an interpreter for programs in \mathcal{L} . Then specialization of the interpreter with respect to a given program $p_{\mathcal{L}}$ is expressed by

$$PS_{\mathcal{K}}([I_{\mathcal{M}}]_{\mathcal{K}}, [[p_{\mathcal{L}}]_{\mathcal{M}}]_{\mathcal{K}})$$

Note that the second argument is a “meta-meta” (or *meta*²) object in the sense that it is the representation in \mathcal{K} of an object already represented in \mathcal{M} .

The result of the first Futamura projection is the representation of a specialized program, say $[I_{\mathcal{M}}^p]_{\mathcal{K}}$. By Definition 6.2.1, $I_{\mathcal{M}}^p(x) = I_{\mathcal{M}}(p, x)$, so $I_{\mathcal{M}}^p$ preserves the functionality of p . $I_{\mathcal{M}}^p$ can be regarded as a “compiled” version of p since it maps input directly to output. This analogy becomes more concrete in practical program specialization systems, which effectively perform the parsing of the object program, leaving residual “execution” operations in the specialized program, as in real compilers.

Thus for instance, we could have $\mathcal{L} = \text{PASCAL}$, $\mathcal{M} = \text{Scheme}$ and $\mathcal{K} = \text{Gödel}$, in which case I is an interpreter for PASCAL written in Scheme, and PS is a specializer of Scheme programs, written in Gödel. The second argument in the first Futamura projection is a Gödel term encoding a Scheme representation of a PASCAL program. The result is a Gödel representation of a Scheme program. Note that to run it as a Scheme program involves extracting the Scheme program from its Gödel representation. In summary, a PASCAL program has been “compiled” into Scheme.

6.3 Self-Applicable Program Specializers

We now consider the case where the program specializer is self-applicable. If $\mathcal{K} = \mathcal{M}$ then in principle the program $PS_{\mathcal{K}}$ can be applied to itself by constructing $[PS_{\mathcal{K}}]_{\mathcal{K}}$. The possibility of encoding a language in itself was established by Gödel numbering, as discussed in Section 4. It is then possible to “self-apply” PS , or more accurately, to apply PS to a representation of itself.

Definition 6.3.1. Second Futamura Projection

The specialization of PS with respect to itself and an interpreter $I_{\mathcal{M}}$ is expressed by

$$PS_{\mathcal{K}}([PS_{\mathcal{K}}]_{\mathcal{K}}, [[I_{\mathcal{M}}]_{\mathcal{K}}]_{\mathcal{K}})$$

Note that the second argument is *meta*². The result, namely $[PS^{I_{\mathcal{M}}}]_{\mathcal{K}}$ is the representation of a program which, when given a program $p_{\mathcal{L}}$ produces $[I_{\mathcal{M}}^p]_{\mathcal{K}}$. This is the representation of a compiled version of $p_{\mathcal{L}}$, as established by the first Futamura projection. Thus the second Futamura projection expresses the production of a compiler from an interpreter for \mathcal{L} .

Definition 6.3.2. Third Futamura Projection

The specialization of PS with respect to itself and (a representation of) itself is expressed by

$$PS_{\mathcal{K}}([PS_{\mathcal{K}}]_{\mathcal{K}}, [[PS_{\mathcal{K}}]_{\mathcal{K}}]_{\mathcal{K}})$$

Again, the second argument is *meta*², and is not the same as the first argument. The result $[PS^{PS}]_{\mathcal{K}}$ is a program which, when given $[I_{\mathcal{M}}]_{\mathcal{K}}$ produces $[PS^{I_{\mathcal{M}}}]_{\mathcal{K}}$. In other words, it returns the representation of a compiler of programs in \mathcal{L} , as established by the second Futamura projection. The third Futamura projection thus expresses the production of a compiler generator (that produces a compiler from a given interpreter).

The second and third projections provide a way of achieving the first projection in stages. This is useful where the same meta-program is to be executed with many different object programs. The “compiler” associated with that meta-program can be obtained using the second projection. If compilers for different meta-programs are to be produced, then the third projection is useful since it shows how to obtain a “compiler-generator” from a partial evaluator.

The effective implementation of the Futamura projections is the subject of current research. The effectiveness of the first Futamura projection is critical to the usefulness of the second and third, which are simply means to achieve the first projection (compilation) by stages. Interpreters, or indeed any meta-programs, do not appear to be any less complex than programs in general from the point of view of specialization. Therefore an effective specializer for the first projection should be also an effective general purpose specializer.

In order to perform the second and third projections the specializer should be effectively self-applicable. This requirement, added to the requirement of being a good general purpose specializer, has proved very difficult to meet. Program analysis methods based on abstract interpretation have been employed to complement partial evaluation and add to its effectiveness (Sestoft & Jones 1988), (Mogensen & Bondorf 1993), (Gurr 1993). Another approach to self-application is to use two or more versions

of a specializer (Ruf & Weise 1993), (Fujita & Furukawa 1988). A simple specializer can be applied to a complex one, or vice versa. In such a method, extra run-time computation in the compiled program or in the compiler produced by the projections is traded for less computation during the second and third Futamura projections respectively.

6.4 Applications of Meta-Program Specialization

The uses of specialization with meta-programming are many, and we finish this section by indicating some areas of current research in which specialization of logic programs is relevant.

Implementing Other Languages and Logics. First-order logic, or fragments of it such as definite clauses or normal clauses, can be used as a meta-language for defining the semantics of other languages and logics. The Futamura projections then offers a general compilation mechanism to improve the computational efficiency of the semantics.

A proof system for a logic \mathcal{L} can be constructed from a set of (abstract) syntax rules for defining expressions of \mathcal{L} , together with a set of inference rules of the form

$$\frac{\alpha_0, \dots, \alpha_k}{\beta}$$

where $\alpha_0, \dots, \alpha_k, \beta$ are expressions in \mathcal{L} , and β is inferred from α_0, \dots , and α_k . Clearly these rules can be encoded as definite clauses, and the concept of theory, proof, theorem and so on can be defined by clauses. The procedural interpretation of clauses can then be used to search for theorems from given object theories. Such an approach may appear naive as a way to build efficient theorem proving systems but, when added to techniques of specialization, practical results are obtainable. The advantage of the method is its generality.

One experiment serves to illustrate this general framework. A theorem prover for first-order clauses, based on the model elimination method, was written as a definite logic meta-program by de Waal and Gallagher (1994). Here, the object language is full clausal logic, while definite clauses are the meta-language. The theorem prover was then specialized with respect to fixed object theories. The result for a given theory was a specialized theorem prover that can prove theorems only in the given theory, but much faster than the original prover. Theory-specific information, such as the usefulness of given object formulas or inference rules in a given proof, can also be obtained and exploited. This can be seen as an application of the first Futamura projection, where the theorem prover is an “interpreter” of clauses.

It appears that this experiment could be repeated for other theorem provers, since the method assumes only that the theorem prover can be

written as a logic meta-program, an assumption that holds for any computable logic. This provides a general approach to using a uniform meta-language (logic programs) to implement other logics, and using specialization to get reasonable efficiency. In similar style, Jones, Gomard, and Sestoft(1993) mention the possibility of using partial evaluation as a way of improving the efficiency of high-level functional meta-languages for programming language definition and implementation.

Expert Systems and Knowledge Based Systems. A deductive data base or knowledge base can be viewed as a logic program. Expert systems can also be included under this heading. Procedures for querying, updating, checking integrity constraints, and similar tasks are thus meta-level procedures. The potential of using partial evaluation to optimize expert system query interpreters with respect to fixed bases of knowledge was identified by Levi and Sardu (1988) and by Sterling and Beer (1989).

Enhanced Language Interpreters. In logic programming, interpreters for tracing computations, spying, timing and so on are sometimes written as “enhanced” versions of a standard or “vanilla” interpreter (see the Proof-Tree interpreter in Figure 4). That is, the standard interpreter is written as a meta-program (usually in the non-ground representation) augmented with operations that record or report the state of the computation at each step. The interpretation overhead is sometimes heavy, but partial evaluation offers a way to reduce this (for a given object program). Safra and Shapiro (1986) report extensive use of this technique for a concurrent logic programming system, where facilities such as deadlock detection were added to a standard interpreter. In their approach the second Futamura projection was constructed by hand since their partial evaluator was not self-applicable.

Optimizing Non-Standard Procedural Semantics. The dual reading of programs, declarative and procedural, represents one of the most distinctive features of logic programming. A program with clear declarative semantics can be regarded as a problem specification. Unfortunately, in order to achieve efficient computations, complex non-standard procedural semantics are often needed. Example of these include coroutining, tabulation and forward checking. The use of such procedural readings is often precluded since they carry a lot of computational overhead. As a result programs are often made more efficient (with respect to standard procedural semantics) at the expense of declarative clarity.

Program specialization appears to offer a general approach to exploiting the dual reading more effectively. The overhead of complex procedural semantics can sometimes be drastically reduced if an interpreter for the semantics is available, written as a meta-program. In this case the Futamura projections can be applied to reduce the overhead. This method was advocated by Gallagher (1986) and illustrated on coroutining programs. Gurr

(1993) also shows the compilation of a corouting example. The work on “compiling control” (Bruynooghe, De Schreye & Krekels 1989) is similar in its aims, but is not based explicitly on meta-programming.

The use of meta-programming combined with specialization as a program production technique is in its infancy. Applications such as those mentioned in this section indicate its generality and promise.

Acknowledgements

We are particularly indebted to John Lloyd and Frank Van Harmelen who have made many suggestions that have improved this chapter. We greatly appreciate their help. We also thank all those who, through discussion and by commenting on earlier drafts contributed to this work. These include Jonas Barklund, Tony Bowers, Henning Christiansen, Yuejun Jiang, Bob Kowalski, Bern Martens, Alberto Pettorossi, Danny De Schreye, Sten-Åke Tärnlund, and Jiwei Wang.

Work on this chapter has been carried out while the first author was supported by a SERC grant GR/H/79862. In addition, the ESPRIT Basic Research Action 3012 (Compulog) and Project 6810 (Compulog 2) have provided opportunities to develop our ideas and understanding of this area by supporting many workshops and meetings and providing the funds to attend.

References

- Abdallah, M. A. N. (1987), Logic programming with ions, *in* O. T., ed., ‘Proceedings of the 14th International Colloquium on Automata, Languages, and Programming, LNCS 267’, pp. 11–20.
- Abramson, H. & Rogers, M., eds (1989), *Meta-Programming in Logic Programming*, MIT Press. Proceedings of the Meta88 Workshop, June 1988.
- Aiello, L. C., Nardi, D. & Schaerf, M. (1988), Reasoning about knowledge and ignorance, *in* ‘Proceedings of the FGCS’, pp. 618–627.
- Barklund, H. & Hamfelt, A. (1994), ‘Hierarchical representation of legal knowledge with meta-programming in logic’, *Journal of Logic Programming* **18**(1), 55–80.
- Barklund, J. (1994), Metaprogramming in logic, Technical Report UP-MAIL Technical Report 80, Department of Computer Science, University of Uppsala, Sweden. to be published in Encyclopedia of Computer Science and Technology, A. Kent and J.G. Williams (eds.), Marcell Dekker, New York, 1994/5.
- Barklund, J., Boberg, K. & Dell’Aqua, P. (1994), A basis for a multilevel metalogic programming language, *in* F. Turini, ed., ‘Proceedings of the

- 4th International Workshop on Meta-Programming in Logic (Meta-94)'. to be published by Springer-Verlag.
- Bowen, K. & Kowalski, R. (1982), Amalgamating language and metalanguage in logic programming, *in* K. Clark & S.-A. Tärnlund, eds, 'Logic Programming', Academic Press, pp. 153–172.
- Bowen, K. & Weinberg, T. (1985), A meta-level extension of Prolog, *in* 'Proceedings of 1985 Symposium on Logic Programming, Boston', pp. 669–675.
- Brogi, A., Mancarella, P., Pedreschi, D. & Turini, F. (1990), Composition operators for logic theories, *in* J. W. Lloyd, ed., 'Computational Logic', Springer-Verlag, pp. 117–134.
- Brogi, A., Mancarella, P., Pedreschi, D. & Turini, F. (1992), Meta for modularising logic programming, *in* A. Pettorossi, ed., 'Proceedings of the Third Workshop on Meta-programming in Logic, Uppsala, Sweden', Springer-Verlag, pp. 105–119.
- Bruynooghe, M., De Schreye, D. & Krekels, B. (1989), 'Compiling control', *Journal of Logic Programming* **6(2–3)**, 135–162.
- Chen, W., Kifer, M. & Warren, D. S. (1993), 'HiLog: A foundation for higher-order logic programming', *Journal of Logic Programming*.
- Christiansen, H. (1994), On proof predicates in logic programming, *in* A. Momigliano & M. Ornaghi, eds, 'Proof-Theoretical Extensions of Logic Programming', CMU, Pittsburgh, PA 15213–3890, USA. Proceedings of an ICLP-94 Post-Conference Workshop.
- Clark, K. L. (1978), Negation as failure, *in* H. Gallaire & J. Minker, eds, 'Logic and Data Bases', Plenum Press, pp. 293–322.
- Clark, K. L. & McCabe, F. G. (1979), The control facilities of IC-PROLOG, *in* D. Michie, ed., 'Expert Systems in the Micro Electronic Age', Edinburgh University Press, pp. 122–149.
- Colmerauer, A., Kanoui, H., Pasero, R. & Roussel, P. (1973), Un système de communication homme-machine en français, Technical report, Groupe d'Intelligence Artificielle, , University d'Aix Marseille II, Luminy, France.
- Costantini, S. (1990), Semantics of a metalogic programming language, *in* M. Bruynooghe, ed., 'Proceedings of the Second Workshop on Meta-programming in Logic, Leuven, Belgium', pp. 3–18.
- Costantini, S. & Lanzarone, G. (1989), A metalogic programming language, *in* G. Levi & M. Martelli, eds, 'Sixth International Conference on Logic Programming, Lisbon', MIT Press, pp. 218–233.
- De Waal, D. & Gallagher, J. (1994), The applicability of logic program analysis and transformation to theorem proving, *in* 'Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy'.

- Dunin-Keplicz, B. (1994), An architecture with multiple meta-levels for the development of correct programs, *in* F. Turini, ed., 'Proceedings of the 4th International Workshop on Meta-Programming in Logic (Meta-94)'. to be published by Springer-Verlag.
- Feferman, S. (1962), 'Transfinite recursive progressions of axiomatic theories', *The Journal of Symbolic Logic* **27**(3), 259–316.
- Fujita, H. & Furukawa, K. (1988), 'A self-applicable partial evaluator and its use in incremental compilation', *New Generation Computing* **6**(2,3), 91–118.
- Futamura, Y. (1971), 'Partial evaluation of computation process - an approach to a compiler-compiler', *Systems, Computers, Controls* **2**(5), 45–50.
- Gallagher, J. (1986), Transforming logic programs by specialising interpreters, *in* 'Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton', pp. 109–122.
- Gallagher, J. (1993), Specialisation of logic programs: A tutorial, *in* 'ACM-SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, Copenhagen', pp. 88–98.
- Gallagher, J. & Bruynooghe, M. (1990), Some low-level source transformations for logic programs, *in* 'Proceedings of Meta90 Workshop on Meta Programming in Logic', Katholieke Universiteit Leuven, Belgium.
- Gardner, P. & Shepherdson, J. (1991), Unfold/fold transformations of logic programs, *in* J.-L. Lassez & G. Plotkin, eds, 'Computational Logic: Essays in Honor of Alan Robinson', MIT Press, pp. 565–583.
- Gödel, K. (1931), 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I', *Monatsh. Math. Phys.* **38**, 173–98. English translation in [Heijenoort 67].
- Gurr, C. (1993), A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel, PhD thesis, Dept. of Computer Science, University of Bristol.
- Harmelen, F. V. (1992), Definable naming relations in metalevel systems, *in* A. Pettorossi, ed., 'Meta-Programming in Logic, Proceedings of the 3rd International Workshop, META-92', Springer-Verlag.
- Hill, P. & Lloyd, J. (1988), Meta-programming for dynamic knowledge bases, Technical Report CS-88-18, Department of Computer Science, University of Bristol.
- Hill, P. & Lloyd, J. (1989), Analysis of meta-programs, *in* H. Abramson & M. Rogers, eds, 'Meta-Programming in Logic Programming', MIT Press, pp. 23–52. Proceedings of the Meta88 Workshop, June 1988.
- Hill, P. & Lloyd, J. (1994), *The Gödel Programming Language*, MIT Press.
- Jiang, Y. (1994), Ambivalent logic as the semantic basis of metalogic programming: I, *in* P. V. Hentenryck, ed., 'Proceedings of the Eleventh

- International Conference on Logic Programming', MIT Press.
- Jones, N., Gomard, C. & Sestoft, P. (1993), *Partial Evaluation and Automatic Software Generation*, Prentice Hall.
- Kanamori, T. & Horiuchi, K. (1987), Construction of logic programs based on generalised unfold/fold rules, in J.-L. Lassez, ed., 'Proceedings of the Fourth International Conference on Logic Programming'.
- Kim, J. S. & Kowalski, R. A. (1990), An application of amalgamated logic to multi-agent belief, in M. Bruynooghe, ed., 'Proceedings of the Second Workshop on Meta-programming in Logic, Leuven, Belgium', pp. 272–283.
- Komorowski, H. (1982), Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog, in '9th ACM Symposium on Principles of Programming Languages; Albuquerque, New Mexico', pp. 255 – 267.
- Kowalski, R. A. (1990), Problems and promises of computational logic, in J. W. Lloyd, ed., 'Computational Logic', Springer-Verlag, pp. 1–36.
- Kowalski, R. A. (1993), Logic without model theory, Technical report, Department of Computing, Imperial College.
- Kursawe, P. (1987), 'How to invent a Prolog machine', *New Generation Computing* **5**, 97–114.
- Levi, G. & Sardu, G. (1988), 'Partial evaluation of metaprograms in a "multiple worlds" logic language', *New Generation Computing* **6(2,3)**, 227–247.
- Lloyd, J. (1987), *Foundations of Logic Programming*, second edn, Springer-Verlag.
- Lloyd, J. & Shepherdson, J. (1991), 'Partial evaluation in logic programming', *The Journal of Logic Programming* **11(3&4)**, 217–242.
- Martens, B. & De Schreye, D. (1992a), A perfect Herbrand semantics for untyped vanilla meta-programming, in K. Apt, ed., 'Proceedings of the Joint International Conference on Logic Programming, Washington, USA', pp. 511–525.
- Martens, B. & De Schreye, D. (1992b), Why untyped non-ground meta-programming is not (much of) a problem, Technical Report CW 159, Department of Computer Science, Katholieke Universiteit Leuven. an abridged version will published in the Journal of Logic Programming.
- Mogensen, T. & Bondorf, A. (1993), Logimix: A self-applicable Partial Evaluator for Prolog, in K.-K. Lau & T. Clement, eds, 'Logic Program Synthesis and Transformation, Manchester 1992', Springer Workshops in Computing.
- Nilsson, U. (1993), 'Towards a methodology for the design of abstract machines for logic programming languages', *Journal of Logic Programming* **161 & 2**, 163–189.

- Pereira, L., Pereira, F. & Warren, D. H. D. (1978), User's guide to DECsystem-10 Prolog, Technical report, Department of A.I., University of Edinburgh.
- Perlis, D. & Subrahmanian, V. S. (1994), Meta-languages, reflection principles, and self-reference, in D. Gabbay, C. Hogger & J. Robinson, eds, 'Handbook of Logic in Artificial Intelligence and Logic programming, Volume II: Deduction Methodologies', Oxford University Press.
- Quine, W. V. O. (1951), *Mathematical Logic (Revised Edition)*, Harvard University Press.
- Richards, B. (1974), 'A point of self-reference', *Synthese*.
- Ruf, E. & Weise, D. (1993), 'On the specialization of online program specializers', *Journal of Functional Programming* **33**, 251–281.
- Safra, S. & Shapiro, E. (1986), Meta interpreters for real, in H.-J. Kugler, ed., 'Information Processing 86', North Holland, pp. 271–278.
- Seki, H. (1989), Unfold/Fold Transformation of Stratified Programs, in G. Levi & M. Martelli, eds, 'Sixth Conference on Logic Programming, Lisbon, Portugal', The MIT Press.
- Sestoft, P. & Jones, N. (1988), The structure of a self-applicable partial evaluator, in H. Ganzinger & N. Jones, eds, 'Programs as Data Objects', North-Holland.
- Shapiro, E. (1982), *Algorithmic Program Debugging*, MIT Press. An ACM Distinguished Dissertation.
- Shepherdson, J. (1994), Negation as failure: Completion and stratification, in D. Gabbay, C. Hogger & J. Robinson, eds, 'Handbook of Logic in Artificial Intelligence and Logic programming, Volume V: Logic Programming', Oxford University Press, chapter 5.
- Sterling, L. & Beer, R. (1986), Incremental flavor-mixing of meta-interpreters for expert system construction, in 'Proceedings of the Third Symposium on Logic Programming, Salt Lake City', pp. 20–27.
- Sterling, L. & Beer, R. (1989), 'Metaintepreters for expert system construction', *Journal of Logic Programming* **6**, 163–178.
- Sterling, L. & Shapiro, E. (1986), *The Art of Prolog*, MIT Press.
- Takeuchi, A. & Furukawa, K. (1986), Partial evaluation of Prolog programs and its application to meta programming, in H.-J. Kugler, ed., 'Information Processing 86, Dublin', North Holland, pp. 415–420.
- Tamaki, H. & Sato, T. (1984), Unfold/fold transformations of logic programs, in 'Proceedings of the Second International Conference on Logic Programming, Uppsala', pp. 127–138.
- Weyhrauch, R. (1980), 'Prolegomena to a theory of mechanised formal reasoning', *Artificial Intelligence* **13**, 133–170.

Weyhrauch, R. (1982), An example of FOL using Metatheory. Formalizing reasoning systems and introducing derived inference rules, *in* 'Proc. 6th Conference on Automatic Deduction'.