J. Expt. Theor. Artif. Intell. 6(1994)239–287.

A metalogic programming approach: language, semantics and applications

STEFANIA COSTANTINI† and GAETANO AURELIO LANZARONE‡

Università degli Studi di Milano, Departimento di Scienze dell' Informazione, Via Comelico 39/41, I-20135, Milano, Italy
†e-mail: costanti@imiucca.csi.unimi.it
‡e-mail: lanzarone@hermesmc.dsi.unimi.it

Abstract. This paper presents a logic programming language of novel conception, called Reflective Prolog, which allows declarative metaknowledge representation and metareasoning. The language is defined by augmenting pure Prolog (Horn clauses) with capabilities of self-reference and logical reflection. Self-reference is designed as a quotation device (a carefully defined naming relation) which allows the construction of metalevel terms that refer to object-level terms and atoms. Logical reflection is designed as an unquotation mechanism (a distinguished truth predicate) which relates names to what is named, thus extending the meaning of domain predicates. The reflection mechanism is embodied in an extended resolution procedure which automatically switches the context between levels. This implicit reflection relieves the programmer from having to explicitly deal with control aspects of the inference process. The declarative semantics of a Reflective Prolog definite program P is provided in terms of the least reflective Herbrand model of *P*, characterized by means of a suitable mapping defined over the Herbrand interpretations of *P*. The extended resolution is proved sound and complete with respect to the least reflective Herbrand model. By illustrating Reflective Prolog solutions to an organic set of problems, and by discussing the main differences with respect to other approaches to logic metaprogramming, we show that the proposed language deploys, within its field of action, greater expressive and inferential power than those available till now. The interpreter of the language has been fully implemented. Because of its enhanced power, logic semantics and working interpreter, Reflective Prolog is offered as a contribution toward making the declarative approach of logic programming applicable to the development of increasingly sophisticated knowledge-based systems.

Keywords: logic programming, reflection, declarative knowledge representation languages, automated reasoning

Received 6 November 1992; accepted 2 January 1993

1. Introduction

The broad *desiderata* of languages for language representation and reasoning,

namely: (i) naturalness and expressivity; (ii) formality, clear semantics; (iii) computational tractability, are notoriously hard to pursue together, and even harder to achieve.

First-order logic is universally recognized as the best knowledge representation formalism w.r.t. (ii), but has been questioned w.r.t. (i). Logic programming, which is based on the Horn-clause subset of first-order logic, on the resolution principle as a uniform proof procedure, and on the least Herbrand model semantics, is considered to be one of the best compromises, currently available, between (ii) and (iii); *a fortiori*, however, it inherits the limitations of first-order logic, whatever they are considered to be.

Several directions have been investigated in order to improve the expressive power of classical logic. Metatheoretical approaches rely on the idea that a formal language with high expressive power (close to the one of a natural language) has to have a self-referential ability (as natural languages have). This means (Perlis 1985) that the language has expressions naming (quoting) its own expressions, and thus can be reasoned about in the language itself. Self-reference implies the need to relate names to what is named (un-naming or un-quoting), hence leading to representation of truth (Perlis 1985, Perlis 1988). Given a language L

$$\text{`}\alpha\text{'} \text{ means } \alpha$$

might be expressed as

$$\text{True}(\text{`}\alpha\text{'}) \text{ iff } \alpha$$

where True is a predicate letter of L, α a wff and 'α' its name. In first-order logic, an equivalent formulation (where T is a theory, $\vdash$ is the provability relation of T and Theorem is, again, a predicate symbol of L):

$$T \vdash \text{Theorem } (\text{`}\alpha\text{'}) \text{ iff } T \vdash \alpha$$

or also, in the form of two inference rules:

$$\text{Theorem } (\text{`}\alpha\text{'}) \mid \alpha, \quad \text{and } \alpha \mid \text{Theorem } (\text{`}\alpha\text{'})$$

These rules, connecting object and meta levels, have been called attachment, or linking, or reflection rules. Weyhrauch (1980) first showed a way to use them in first-order logic, and Bowen and Kowalski (1982) introduced them in logic programs in order to enhance their representational and reasoning capabilities.

In the development of artificial intelligence systems, metalevel formulations are ubiquitous, in that they have been used in a number of domains and with a wide variety of purposes and architetures (see Aiello *et al.* 1986, van Harmelen 1989 for a comprehensive overview and classification, respectively). Recognized advantages of metalevel representations are in the possibility of separation between domain knowledge and control knowledge, and of better mastering of inference by explicit treatment of control. Other important advantages of using metaknowledge and metareasoning in knowledge-based systems have been recently acknowledged (see Benjamin 1990 for a discussion). Metatheoretic concepts are suitable to express knowledge about how to perform generalizations, or about problem reformulation, or about inductive biases. More generally, they permit the concise statement of generalizations that are useful in problem-solving (Benjamin 1990).

In the field of logic programming, after the original proposal by Bowen and Kowalski (1982), metaprogramming has become a common technique for software development in Prolog (a survey of the main current approaches to metaprogramming in logic programming can be found in Abramson and Rogers 1989, Bruynooghe 1990). The metalinguistic constructs of Prolog on the one hand increase the expressive power of Horn clauses, make them usable as a practical programming language, and are necessary to metaprogramming. On the other hand, they are low-level, have no logical semantics and have no predefined relationship with the object-level language. Thus, metaprograms have only a procedural, often unduly complicated, semantics (a critical analysis of Prolog metaprograms is given in Hill and Lloyd 1988). It is only recently that new independent proposals (Hill and Lloyd 1988, Subrahmanian 1988) have been presented to provide a theoretical foundation for traditional Prolog metaprogramming.

In this paper, we present a self-referential Horn-clause language with logical reflection, called Reflective Prolog (RP for short). Since it stays within the framework of logic programming and is intended for declarative representation of metaknowledge and metareasoning, we call this approach 'metalogic programming' (this denomination was introduced in Costantini and Lanzarone 1988, and, independently and in a different context, in Subrahmanian 1988). The objective of this approach is that of developing a more expressive and powerful language, while preserving the essential features of logic programming: Horn-clause syntax, model-theoretic declarative semantics, resolution via unification as procedural semantics, correctness and completeness properties. Clearly, we do not mean that RP is more powerful as a formal system (since Horn clauses are by themselves Turing complete (Tarnlund 1977)) but rather as a knowledge programming language: 'One way to judge the usefulness of a language is by its ability to permit concise statements useful in solving an interesting class of problems, or in answering an interesting class of queries' (Pylyshym 1984). Also, we do not claim that RP's metalevel capabilities are entirely new *per se*, but we do maintain that it offers a novel combination of these capabilities, yielding an original balance among the three broad *desiderata* mentioned at the beginning of this section.

The basic rationale underlying the design of RP is as follows. Horn clauses with resolution are taken as the core of the language, because of their computational tractability. Horn clauses are extended with self-reference, and resolution is extended with logical reflection, in order to achieve greater expressive and inference power. The logical reflection of RP is conceptually based on the idea of reflection principle as originally introduced in the context of symbolic logic by Feferman (1972), where it was intended as 'the description of a procedure for adding to any set of axioms A certain new axioms whose validity follows from the valdity of the axioms A and which formally express, in the language of A, evident consequences of the assumption that all the theorems of A are valid'. As Feferman pointed out, adding a reflection principle to a theory may lead to conservative or non-conservative extensions, and/or to different kinds of extensions. Reflective Prolog's main characteristics are the following.

- Language and metalanguage are amalgamated in a non-conservative extension. This means that statements are provable in the amalgamated language, that

are provable neither in the language nor in the metalanguage alone. This is a different and more powerful combination w.r.t. the FOL system (Weyhrauch 1980) and the Goedel language (Hill and Lloyd 1991), which are not amalgamated, and also w.r.t. Bowen–Kowalski's amalgamation (Bowen and Kowalski 1982), which is conservative.

- The reflection mechanism is implicit, i.e. the interpreter of the language automatically reflects upwards and downwards on occurrence of predefined conditions. This allows reasoning and metareasoning to interleave without the user's intervention, contrary to the systems of Weyhrauch 1980, Smith 1984, Bowen and Kowalski, 1982, where reflection is explicit, i.e. has to be specified in advance in the program.
- The reflection principle is embedded in both the procedural and the declarative semantics of the language, that is, in the extended resolution procedure which is used by the interpreter, and in the construction of the models which give meanings to programs. Procedurally, this implies that axiomatization of standard provability in the metatheory (required in Bowen and Kowalski 1982) is not needed. Object-level reasoning is not simulated by meta-interpreters but directly executed by the language interpreter, thus avoiding unnecessary inefficiency. Semantically, a device closer to Feferman's reflection principles rather than to Bowen–Kowalski's linking rules is introduced to characterize the models of a Reflective Prolog program. The formal semantics is defined hands-in-hands with the behaviour of the interpreter, as opposed for instance to MetaProlog (Bowen and Weinberg 1985), where provability is hardwired in the interpreter but the semantics was not defined until much later (Subrahmanian 1988).
- The language achieves a substantial part of self-referential first-order logic advocated in Perlis (1985), Davies (1990) (where implementability was not at issue). It obtains higher-order features in a first-order language with first-order syntax, differently from higher-order extensions of Prolog, like λProlog (Miller and Nadathur 1988), or HiLog (Chen *et al.* 1989).

In this paper we present the syntax, the declarative and procedural semantics, and sample applications of Reflective Prolog. Declarative semantics is defined in a model-theoretic fashion, providing a least model semantics based on a notion of the *reflective Herbrand model* of a theory. The least reflective Herbrand model is characterized (similarly to the classical Horn-clause language semantics (Lloyd 1987)) as the least fixpoint of a suitable mapping, in order to provide a link between the declarative and procedural semantics of a program. Derivation by resolution is extended to include forms of implicit reflection to switch between levels. Extended resolution is proved sound and complete with respect to the least reflective Herbrand model of a program. This semantics is not a departure from the classical semantics of logic programs; rather, it is an extension, based on the same approach and enjoying the same properties.

The paper is organized as follows. In sections 2 and 3, the formal syntax and semantics of Reflective Prolog are introduced, and the proofs of the achieved results are presented. In section 4, the extended unification and resolution are formally defined. In section 5, a set of sample applications of the language is presented in detail. In section 6, Reflective Prolog is compared with Prolog meta-interpreters, and the question of how computational efficiency is affected

by the proposed extensions and possible ways to cope with it are also discussed. In section 7, directions of continuation of this research are outlined; in particular, the smooth introduction in RP of a powerful form of (metalevel) negation based on a similar reflection principle is mentioned. Finally, the results on soundness and completeness of the extended resolution are reported in the Appendix.

2. Syntax

In this and in the next sections, we take as base reference for the foundations of logic programming the comprehensive presentation of Lloyd (1987), to which in the following we implicitly refer when saying 'as usual'.

2.1. *Alphabet*

The main difference in the alphabet of RP w.r.t. traditional Horn-clause languages is the presence of *metavariables* and *name constants* in addition to *object variables* and *object constants*. There are three kinds of metavariables, called *function* metavariables, *predicate* metavariables and *general* metavariables, and three kinds of name constants: *quoted* name constants, and two kinds of *bracketed* name constants.

Definition 2.1. The alphabet Π of a Reflective Prolog program P is a union of the following (disjoint) sets:

- A (countably infinite) set of object variables. Each variable is written as a sequence of characters in which the first is uppercase alphabetic.
- A (countably infinite) set of metavariables. Each metavariable is written as a sequence of characters in which the first is either the character '%' (function metavariable) or the character '#' (predicate metavariable) or the character '$' (general metavariable).
- A (finite, possibly empty) set of *function symbols*. It contains all the function symbols appearing in P, which are written as sequences of characters where the first is lowercase alphabetic.
- A (finite, non-empty) set of *predicate symbols*. It contains all the predicate symbols appearing in P, which are written as sequences of characters where the first is lowercase alphabetic.
- The set of the distinguished function symbols predication, predicate, function, functor, arity, args.
- A (finite, non-empty) set of object constants. It contains all the object constants appearing in P (if there are none, we add one to the alphabet), which are written as sequences of characters where the first is lowercase alphabetic.
- A (countably infinite) set of name constants. Each name constant is written as a sequence of characters included either in quotation marks ("...") (then called *quoted* name constant), or in angle brackets (⟨...⟩) (then called *predicate* name constants), or in curly brackets ({...}) (then called *function* name constant). This set is the union of the following sets:
  - — the finite (possibly empty) set of the name constants appearing in P;
  - — the finite set of the predicate name constants corresponding to the predicate symbols appearing in P (i.e., for each predicate symbol p, the constant ⟨p⟩);

— the finite set of the function name constants corresponding to the function symbols appearing in P (i.e. for each function symbol f, the constant {f});
— the finite set of the quoted name constants corresponding to the object constants appearing in P (i.e. for each object constant c, the constant "c");
— the countably infinite set of the quoted name constants corresponding to all the symbols above (i.e. for each symbol s, the constants "s", ""s"", and so on).

- The distinguished predicate symbols solve (unary), theory_clause (binary), ref (binary).
- A set of *operators* (connectives and punctuation symbols), viz. ":-", ",", ".", "(",")", "[","]".■

2.2. *The language of a program*

The language defined by a Reflective Prolog program P consists of all the well-formed formulae in definite clausal form obtained from the alphabet of P. The definition of terms is extended to include both object variables and metavariables, and a new class of terms, *name terms*.

Definition 2.2. Name terms are defined inductively as follows:

- Name constants (either quoted or bracketed), described above.
- Metavariables (either function or predicate or general), described above.
- Function name terms, which are of the form function(functor(ϕ), arity(n), args([α_1, ..., α_n])), where ϕ is a function name constant or a function metavariable or a general metavariable, n is a natural number and α_1, ..., α_n are name terms.
- Relation name terms, which are of the form predication(predicate(ρ), arity(n), args([α_1, ..., α_n])), where ρ is a predicate name constant or a predicate metavariable or a general metavariable, n is a natural number and α_1, ..., α_n are name terms.■

Definition 2.3. Terms are defined inductively as follows:

- Object variables, described above.
- Object constants, described above.
- Name terms, described above.
- Lists, which are of the form [τ_1, ..., τ_n] ($n \geq 0$), where τ_1, ..., τ_n, are terms.
- Functional applications, of the form f(τ_1, ..., τ_n), where f is an n-ary function symbol and τ_1, ..., τ_n are terms.■

Definition 2.4. Reflective Prolog atomic formulae (*atoms* for short) are of the form p(τ_1, ..., τ_k), where p is a predicate symbol and τ_1, ..., τ_k are terms.■

Terms can be seen as being divided into *metalevel terms*, that contain name terms as subterms, and *object-level terms*, that do not. *Metalevel atoms* contain at least one metalevel term as argument, *object-level atoms* do not.

Name terms allow the representation of language entities in the language itself.

Quoted name constants are intended as names for constant symbols, and bracketed name constants as names for predicate and function symbols. Compound name terms, which are built out of distinguished function symbols, are intended as names for other terms (function name terms), and atoms (relation name terms). For example, the name of the term f(a) is the term

```
function(functor({f}), arity(1), args(["a"])),                    (2.1)
```

the name of the atom p(X) is the term

```
predication(predicate(⟨p⟩), arity(1), args(["X"])),               (2.2)
```

and the name of the atom q(f(a),⟨b⟩) is the term

```
predication(predicate(⟨q⟩), arity(2),                             (2.3)
        args([function(functor({f}), arity(1), args(["a"])), "⟨b⟩"])).
```

Names of names are also possible: the name of (2.2) (i.e. the name of the name of the atom p(X)) is

```
function(functor({predication}), arity(3),
        args([function(functor({predicate}), arity(1),args(["⟨p⟩"])),
            function(functor({arity}),arity(1),args(["1"])),      (2.4)
            function(functor({args}),arity(1),args([""X""])])])
```

In a similar way it is possible to build names of names of names, and so on.

The Reflective Prolog programmer is not forced to use such an awkward notation: a shortened form is allowed, where distinguished function symbols (even when quoted one or more times) and arity are omitted. Namely, (2.1)–(2.4) above can be shortened as:

```
{f}("a")                                                          (2.1′)
⟨p⟩("X")                                                          (2.2′)
⟨q⟩({f}("a"),"⟨b⟩")                                               (2.3′)
"⟨p⟩"(""X"").                                                     (2.4′)
```

Conversion between the two forms is taken care of by the language parser. The short form is clearly most commonly used in actual programs. The reasons for such a naming device are discussed in the following sections.

Notice that the arguments of metalevel terms or atoms need not be at the same level of quotation. For instance, pred({f}("a"),⟨p⟩,g(""b""),h(X)), where pred and p are predicate symbols and f, g, h are function symbols, is an allowed metalevel atom.

2.3. *Definite programs*

Reflective Prolog formulae are in definite clausal form, as usually defined. Thus, we have:

- *unit clauses* (facts), which are of the form α. where α is an atom.
- *non-unit clauses* (rules), which are of the form α:–β. (or, equivalently, $\alpha \leftarrow \beta$.), where α is an atom and β a conjunction (of the form $\alpha_1, \ldots, \alpha_n$, where $\alpha_1, \ldots, \alpha_n$ are atoms).

- *goals* (queries), which are of the form ?-β. (equivalently, either $\leftarrow\beta$. or :-β.), where β is a conjunction.

The definition of Reflective Prolog *definite clause* includes, however, some syntactic restrictions. First, there are three kinds of clauses: object-level clauses, where the conclusion as well as the conditions are object-level atoms; metalevel clauses, where the conclusion is a metalevel atom, and the conditions are either metalevel atoms or object-level atoms; meta-evaluation clauses, which are metalevel clauses containing one or more atoms with solve as predicate symbol. Furthermore, use and mention of a predicate in the same clause are not allowed: i.e. ⟨p⟩ cannot appear in a clause containing an atom p(...). The argument of solve must be a relation name term representing a goal; since atoms such as solve(⟨solve⟩(...)) are forbidden, being a special case of use and mention of a predicate in the same clause, then meta-meta-evaluation is not allowed in Reflective Prolog. Finally, the distinguished predicates ref and theory_clause are predefined, and thus cannot appear in the head of clauses.

Definition 2.5. A *safe-referential* clause is a definite clause where, for each composing atom B_j, $j \in \{1, \ldots, n\}$, the following conditions hold:

(i) if $B_j = p_j(a_1, \ldots, a_k)$, then $\forall\, i \in \{1, \ldots, k\}\ \forall\, g \in \{1, \ldots, n\}$ a_i does not contain $\langle p_g \rangle$ as subterm;
(ii) if $B_j = \text{solve}(\alpha)$, then α is a relation name term;
(iii) if B_j is the clause head, then $B_j \neq \text{ref}(\ldots), \text{theory\_clause}(\ldots)$.■

Definition 2.6. A *meta-evaluation predicate* p is a predicate which is directly or indirectly used in the definition of solve, i.e. it either appears in the body of a clause whose head is solve(...), or is directly or indirectly used in the definition of some other predicate appearing in the body of such a clause.■

Definition 2.7. A Reflective Prolog program clause is a safe-referential unit clause, or a safe-referential non-unit clause A :-$A_1, \ldots, A_n$ for which one of the following conditions holds:

(i) the conclusion A and the conditions $A_1, \ldots, A_n$ are object-level atoms (object level clause);
(ii) the conclusion A is a metalevel atom, and the conditions $A_1, \ldots, A_n$ are either metalevel atoms or object-level atoms (metalevel clause);
(iii) the same as (ii), where A=solve(...)
(meta-evaluation clause, specifically solve clause)
(iv) the same as (ii) where A=p(...) with p meta-evaluation predicate, and $\exists\, j \in \{1, \ldots, n\} : A_j = \text{solve}(\alpha)$ (meta-evaluation clause).■

A Reflective Prolog *definite program*, that is a finite set of program clauses, can be seen as being divided into two main levels. The first one, called *meta-evaluation level*, consists of meta-evaluation clauses, and in practice contains the definition of the distinguished predicate solve and of its auxiliary predicates (i.e. the meta-evaluation predicates of definition 2.6). The second level, called *base level*, consists of the remaining metalevel clauses and of all the object-level clauses.

The distinction between base level and meta-evaluation level is related to the choice of adopting implicit rather than explicit reflection in the language. The reasons of this choice and the importance of the syntactic distinction are discussed in the rest of this paper. The point stated in definitions 2.6 and 2.7 is that the predicate solve cannot be, in general, explicitly used in clauses. It is however necessary to allow both direct and indirect recursion in the definition of solve. Direct recursion is coped with in case (iii) of definition 2.7. Indirect recursion in definition 2.6 and case (iv) of definition 2.7.

Definition 2.8. A Reflective Prolog definite program is a finite set of Reflective Prolog definite program clauses.■

Definition 2.9. A Reflective Prolog definite goal is a safe-referential clause of the form :-B_1, ..., B_n.■

Example 1. The following is a Reflective Prolog definite program.

```
/* metaevaluation level */
[1] solve(#P($X, $Y)) :- symmetric(#P), solve(#P($Y, $X)).
[2] solve(#P($X, $Y)) :- equivalent(#P, #Q), solve(#Q($X, $Y)).

/* base level */
    /* metalevels */
    symmetric(⟨friend⟩).
    symmetric(⟨equivalent⟩).
[3]   equivalent(⟨amico⟩,⟨friend⟩).
      /* object level */
      friend(giorgio, mary).
      amico(lucy, albert).
      happy(X):-friend(X,lucy).
```

The base level consists of object-level facts and rules, defining the relations friend, amico and happy, as well as metalevel facts. The metalevel fact [3] states that the relations amico and friend are 'equivalent' (in the sense that their names are one the translation of the other). Notice that the relation equivalent is defined over bracketed name constants, which act as names for predicate symbols: the choice of special constants to name predicates and functions (instead of function symbols like, for instance, in Hill and Lloyd 1988) aims at making metalevel clauses involving predicate names easier to write and understand. The relations friend and equivalent are both asserted to be symmetric.

The two solve rules constitute the meta-evaluation level for the base-level theory: rule [1] declaratively defines the meaning of symmetry in the theory, stating that (the objects whose name is denoted by) $X and $Y are in the relation (whose name is denoted by) #P, provided that #P is asserted to be symmetric and that $Y and $X are in the relation #P. Rule [2] states that equivalent relations have the same extension.■

Meta-evaluation rules can be considered from different points of view:

- as declarative definition of the intended meaning of base-level relations;
- as connecting intensional properties of relations and functions (like symmetry and equivalence in the example) to their extensional counterpart; in particular,

they relate properties expressed on names to the actual extension of the named entities;
- as auxiliary inference rules, with both a declarative effect (increasing, as we will see, the set of consequences of the program) and a procedural effect (allowing the derivation of more facts).

3. Declarative semantics

The declarative semantics of a Reflective Prolog definite program P is defined by means of a notion of interpretation that extends the usual one to deal with the metalevel part of the language. We start with the definition of the *naming relation*, that relates name terms to the linguistic entities they are intended to refer to, and with the definition of a *pre-interpretation*.

3.1. *Naming relation and pre-interpretation*

The *extended Herbrand universe* U_E, defined as usual (Lloyd 1987), contains (by the definition of the language of P) object terms as well as name terms and metalevel terms. Let $NT \subset U_E$ be the set of all the name terms belonging to U_E, and $RNT \subseteq NT$ the set of all the relation name terms. The *extended Herbrand base* B_{PE} is defined as usual, starting from U_E.

We now introduce the naming relation NR such that α NR β means that α is the name of β. By a slight abuse of notation, we also write $(\alpha_1, \ldots, \alpha_n)$ NR $(\beta_1, \ldots, \beta_n)$, meaning $(\alpha_1$ NR $\beta_1) \wedge \ldots \wedge (\alpha_n$ NR $\beta_n)$. In order to define NR, we introduce the set $\Lambda = U_e \cup \Pi$, where Π is the alphabet of the language of P. Λ contains all the terms of the language of P, plus those symbols of the alphabet (namely, function and predicate symbols) that have a name, but are not terms of the language.

Definition 3.1. The naming relation $NR \subseteq NT \times \Lambda$ is defined as follows.

1. Each quoted name constant is NR-related to the symbol enclosed in the quotes.
2. Each bracketed name constant is NR-related to the symbol in the brackets.
3. If $\langle p \rangle$ NR p and $(\alpha_1, \ldots, \alpha_n)$ NR $(a_1, \ldots, a_n)$ then we have
   predication(predicate($\langle p \rangle$),arity(n),args($[\alpha_1, \ldots, \alpha_n]$)) NR $p(a_1, \ldots, a_n)$
   i.e. a relation name term is the name of an atom.
4. If {f} NR f and $(\alpha_1, \ldots, \alpha_n)$ NR $(a_1, \ldots, a_n r)$ then we have
   function(functor({f}),arity(n),args($[\alpha_1, \ldots, \alpha_n]$)) NR $f(a_1, \ldots, a_n)$
   i.e. a function name term is the name of a term. ■

With respect to the examples in section 2.2, NR relates (2.1′) to the term f(a), (2.2′) to the atom p(X), (2.3′) to the atom q(f(a),⟨b⟩) and (2.4′), which is the name of a name, to the name term ⟨p⟩("X").

Given the naming relation NR and given an element A of Λ, we adopt the notation $\uparrow$A to indicate the name term α such that α NR A. We also write $\alpha = \uparrow$A (or, conversely, A = $\downarrow \alpha$) meaning α NR A (conversely, A NR^{-1} α). For instance, $\uparrow$p(a) is the relation name term ⟨p⟩("a") if p is a predicate symbol, or the function name term {p}("a") if is p a function symbol; $\downarrow$⟨q⟩({f}("X")) is the atom q(f(X)); $\downarrow$"⟨h⟩(""a"")" is the relation name term ⟨h⟩("a"). The application of NR will be called *dereferentiation*, and the application of its inverse NR^{-1}

referentiation. Notice that, according to the above definition, referentiation is possible on every term, while dereferentiation is possible on name terms only. The naming relation between $\uparrow$A and A is made explicit in Reflective Prolog, provided that A belongs to U_E: the predefined binary predicate ref(α,A) is true if $\alpha = \uparrow$A.

The definition of a pre-interpretation PI_E over U_E is a simple extension of the usual notion.

Definition 3.2. A Pre-Interpretation PI_E over U_E consists of the following assignments.

[1] Each constant is assigned to itself in U_E.
[2] Each n-ary function symbol f is assigned a mapping

$$PI_E(f) : U_E^n \rightarrow U_E$$

that maps $a_1, \ldots, a_n$ over $f(a_1, \ldots, a_n)$.
[3] Each list $[\tau_1, \ldots, \tau_n]$ is assigned the sequence $[PI_E(\tau_1), \ldots, PI_E(\tau_n)]$.■

The definition of pre-interpretation is independent of NR. Consequently, each metalevel term (e.g. f(⟨p⟩)) is pre-interpreted onto itself in U_E, regardless of what the contained name terms are intended to represent. As we will show later, the definition of interpretation relates the named objects and their metalevel properties (e.g. the extension of the predicate p and the property expressed by f) via reflection.

3.2. *Variable and term assignment*

While the notion of *term assignment* is the usual one, we need to redefine the notion of *variable assignment* to deal with the different types of metalevel variables.

Definition 3.3. Let PI_E be a pre-interpretation. A variable assignment (w.r.t. PI_E) is an assignment to each variable of an element of the domain U_E of PI_E, such that:

(a) the assignment of each object variable is an object-level term;
(b) the assignment of each predicate metavariable is a predicate name constant;
(c) the assignment of each function metavariable is a function name constant;
(d) the assignment of each general metavariable is a metalevel term.■

It is worth noting that Reflective Prolog is actually a typed language: object variables range over object terms, predicate metavariables over predicate name constants (i.e. over names of predicates), function metavariables over function name constants (i.e. over names of functions), and general metavariables over metalevel terms in general. This simple kind of typing is mainly aimed at avoiding, as far as possible, the use of explicit recognizers. For instance, in order to define at the metalevel a class of predicates, it is sufficient to give a predicate metavariable as argument to the relation representing that class, instead of explicitly checking whether it is the name of a predicate. In order to define the language semantics, however, it is sufficient to deal with types with respect to variable assignment and unification only.

An *instance* of a term or atom τ is the term or atom τ' obtained from τ by applying a variable assignment. Similarly, an instance of a clause ω is the clause ω' obtained from ω by applying the same variable assignment to all the composing atoms.

3.3. *Extended Herbrand interpretation*

Finally, we introduce the notion of *extended Herbrand interpretation* I_E of the language defined by a Reflective Prolog definite program P, based on a pre-interpretation PI_E, I_E can be characterized, as usual, as a subset of the extended Herbrand base B_{PE} consisting of the set of all ground atoms which are true w.r.t. the interpretation. Thus, $r(a_1, \ldots, a_n) \in I_E$ means that $r(a_1, \ldots, a_n)$ is true w.r.t. I_E.

Let LRNT be the set of all the sequences (of any length) of relation name terms. Let P' be the instantiated version of P, i.e. the program obtained by substituting in every possible way the variables appearing in P by terms in U_E.

Definition 3.4. An extended Herbrand interpretation I_E based on a pre-interpretation PI_E, on the naming relation NR and on P' consists of the following assignments.

[1] The assignment of each non-distinguished n-ary predicate symbol r is a mapping

$$I_E(r) : U_E^n \rightarrow \{\text{true, false}\}$$

where $r(a_1, \ldots, a_n) \rightarrow$ false if $\exists\, i \in \{1, \ldots, n\} : a_i$ NR r (i.e. $a_i = \langle r \rangle$) (auto-mention is forbidden).

[2] The assignment of the distinguished unary predicate symbol solve is a mapping

$$I_E(\text{solve}) : \text{RNT} \rightarrow \{\text{true, false}\}$$

where solve(α) $\rightarrow$ true if $A = \downarrow\alpha \in I_E$, $A \neq$ solve(...).

[3] The assignment of the distinguished binary predicate symbol theory_clause is a mapping

$$I_E(\text{theory\_clause}) : \text{RNT} \times \text{LRNT} \rightarrow \{\text{true, false}\}$$

where theory_clause(α, [β_1, ..., β_n]) $\rightarrow$ true if $A \leftarrow B_1, \ldots, B_n$ or solve(α)$\leftarrow B_1, \ldots, B_n$ is a clause in P', where $\downarrow\alpha = A$ or $\downarrow\alpha =$ solve($\uparrow A$), and $B_1 = \downarrow\beta_1, \ldots, B_n = \downarrow\beta_n$.

[4] The assignment of the distinguished binary predicate symbol ref is a mapping

$$I_E(\text{ref}) : \text{NT} \times U_E \rightarrow \{\text{true, false}\}$$

where ref(α, A) $\rightarrow$ true if $\alpha = \uparrow A$.■

Notice that solve (which is true if the atom named by its argument in NR belongs to the interpretation) not only 'mirrors' the interpretation of predicates, but can also 'extend' it, since it is possible that solve(α) $\in I_E$ while $A = \downarrow\alpha \notin I_E$. The definition of Reflective Models, given below, will clarify how this extension is made effective.

Clearly, an extended Herbrand interpretation is also an interpretation in the

usual sense. Since the converse does not hold, it is possible to identify the set $HE \subseteq 2^{BPE}$ of all the extended Herbrand interpretations of P, which is a complete lattice under the partial order of set inclusion (with the empty set as the bottom element, and B_{PE} as the top element). If the program and the alphabet are non-empty, the bottom element of HE can be assumed to be the interpretation $\perp$ where the only interpreted predicates are the predefined ones theory_clause and ref. That is, $\perp$ characterizes the naming relation, as well as the clauses of the program. It is easy to see (by the definition of an extended Herbrand interpretation) that $\perp$ is uniquely determined by the program *P*, and that it is contained in every other extended Herbrand interpretation.

3.4. *Reflective Models*

The classical notions of model and of logical consequence are extended to the notions of *reflective model* and of *reflective logical consequence.*

Definition 3.5 Let P be a Reflective Prolog definite program and L the language of P. A reflective model for P is an extended Herbrand interpretation I_E for L which is a model for

$$P' = P \cup \{A \leftarrow solve(\uparrow A) : A \in B_{PE}, A \neq solve(\ldots)\}.\blacksquare$$

Reflective models are also Herbrand models in the usual sense (Lloyd 1987), and it can be noted that the model intersection property obviously holds for Reflective Prolog programs. Then there exists a least reflective Herbrand model RM_P of a definite program, which is in general wider than the least Herbrand model.

Definition 3.6 An atom A is a reflective logical consequence of a Reflective Prolog definite program P if, for every interpretation I_E, I_E is a reflective model for P implies that I_E is a model for A.■

The axioms $A \rightarrow solve(\uparrow A)$ will be called in the following *reflection axioms*. Adding the reflection axioms to a program P corresponds to applying to P the *reflection principle* introduced in Feferman (1962), definition 2.16. The purpose here is to extend the usual notion of model, in order to include in the reflective model of a program P those atoms A such that $solve(\uparrow A)$ is a logical consequence of P. In this way, the extensions to the 'intended meaning' of base-level relations (and of their interconnections) defined via metalevel clauses and solve rules have their semantic counterpart.

The notion of extended Herbrand interpretation formalizes the assumption that the meta-evaluation level has total visibility on the base level. We modified the classical semantics of the Horn-clause language on this point since, in our opinion, this is necessary *in any* metalogic approach. The interaction in the opposite direction is instead a design choice of the language. Then, for formalizing the specific design choice of Reflective Prolog we choose to modify the program (by adding the reflection axioms) rather than further modifying the semantics (by defining the extended Herbrand interpretation so as to entail A if and only if $solve(\uparrow A)$). The two possibilities lead to equivalent results in this specific context, but the latter seems too restrictive in principle. In fact, the reflection axioms of Reflective Prolog are meant to express just one of the reflection principles potentially applicable. By keeping the reflection axioms explicit,

extensions and/or modifications to the language can be performed by changing the reflection axioms and/or adding new ones, without any semantic modification.

Notice that the solve predicate is not an axiomatization of provability, but it can be seen as a truth predicate, in the sense mentioned in Perlis (1985). It is also different from Bowen and Kowalski's demo (Bowen and Kowalski 1982), as well as from Prolog meta-interpreters (Sterling and Shapiro 1986) where it is necessary to include rules defining the standard inference strategy. Also, in Bowen and Kowalski (1982) and Subrahmanian (1988) demo("G") is a logical consequence of a program if and only if G is. Instead, solve rules do not provide a conservative extension: the least reflective Herbrand model is in general a significant superset of the usual least Herbrand model.

Example 2. Let us consider the program below, which is a simplification of example 1.

```
solve(#P($X, $Y)):-symmetric(#P),solve(#P($Y, $X)).
symmetric(⟨friend⟩).
friend(lucy,albert).
happy(X):-friend(X,lucy).
```

The traditional least Herbrand model M_P would contain only the two atoms of the two program facts, since they do not correspond to the conditions of any rule.

```
/* Least Herbrand Model MP */
symmetric(⟨friend⟩)
friend(lucy,albert)
```

If we consider extended Herbrand interpretations, where solve($\uparrow$A) $\in I_E$ if A $\in I_E$ (*upward reflection*), M_P becomes (omitting the standard subset $\perp$):

```
/* Least Herbrand Model MP over Extended Herbrand Interpretations */
symmetric(⟨friend⟩)                    (i)
friend(lucy, albert)                   (ii)
solve(⟨symmetric⟩("⟨friend⟩"))         (iii)
solve(⟨friend⟩("lucy", "albert"))      (iv)
solve(⟨friend⟩("albert", "lucy"))      (v)
```

Specifically, atoms (iii) and (iv) 'mirror' (i) and (ii); but, now, atoms (i) and (iv) correspond to the conditions of the solve rule, and then a model will contain its conclusion, namely the atom (v).

If we now look for the least reflective Herbrand model (by augmenting the program via the reflection axioms A $\leftarrow$ solve($\uparrow$A)), the reflection axiom with (v) as condition leads to add the conclusion, namely the atom (vi) below (*downward reflection*); but in turn (vi) corresponds to the condition of the base-level rule, thus leading to add the new atoms (vii) and (viii). This is an example of *interleaving* between levels.

```
.../* RMP: the same as above, plus: */
friend(albert, lucy)                   (vi)
happy(albert)                          (vii)
solve(⟨happy⟩("albert"))               (viii)
```

Clearly, RM_P is not minimal in the traditional sense: it is however minimal if we

intend to capture the meaning of solve clauses considered as auxiliary inference rules.■

3.5. *Fixpoint semantics*

The declarative semantics of a Reflective Prolog definite program P can be defined as the fixpoint of a mapping defined over the set HE of all the extended Herbrand interpretations of P. This mapping, called T_{PE}, can be seen as an extension of the mapping T_P defined in Lloyd (1987).

Definition 3.7. Let P be a Reflective Prolog definite program. The mapping $T_{PE} : HE \rightarrow HE$ is defined as follows.

$T_{PE}(I_E) = \{A, S \in B_{PE}:$
$A \leftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P,
$\{A_1, \ldots, A_n\} \subseteq I_E$, $A \neq solve(\alpha)$,
[1] $S = solve(\uparrow A)\ \} \cup$
$\{S, A \in B_{PE}:$
$S \leftarrow S_1, \ldots, S_n$ is a ground instance of a clause in P,
$(S_1, \ldots, S_n\} \subseteq I_E$,
[2] $S = solve(\uparrow A)\ \}$.■

Proposition 3.1. Let $I_E \in HE$. Then $T_{PE}(I_E) \in HE$ (that is, T_{PE} maps extended Herbrand interpretations onto extended Herbrand interpretations.)

Proof. Straightforward.■

It can be noted that the definition of T_{PE} subsumes the linking rules of Weyhrauch (1980) and Bowen and Kowalski (1982). In fact, including the atom S ([1]) in the result of the application of T_{PE} corresponds to communicating results from the base level to the meta-evaluation level (thus modelling upward reflection); vice versa, the inclusion of A ([2]) means that results are reflected from the meta-evaluation level to the base level (thus modelling downward reflection). But while in the mentioned approaches a proof is performed at a certain level (object or meta) and then results can be communicated to the other level, in subsequent applications of T_{PE} the switching between base and meta-evaluation level is performed on single subgoals (interleaving between levels), thus exploiting all the useful intermediate results no matter the level where they were obtained.

The mapping T_{PE} is clearly monotonic, and we have the following propositions.

Proposition 3.2. Let *P* be a Reflective Prolog definite program. Then the mapping T_{PE} is continuous.

Proof. Let X be a directed subject of HE. Note first that $\{A_1 \ldots A_n\} \subseteq lub(X)$ iff $\{A_1 \ldots A_n\} \subseteq I$, for some $I \in X$. In order to show that T_{PE} is continuous, we have to show that $T_{PE}(lub(X)) = lub(T_{PE}(X))$ for each directed subset X. Now we have that $A \in T_{PE}(lub(X))$ iff one of the points (i)–(iii) holds:

(i) $A \leftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq lub(X)$
iff $A \leftarrow A_1, \ldots A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq I$, for some $I \in X$
iff $A \in T_{PE}(I)$, for some $I \in X$
iff $A \in lub(T_{PE}(X))$

(ii) $A = solve(\uparrow B), B \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P and $\{B_1, \ldots, B_n\} \subseteq lub(X)$
iff $B \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P and $\{B_1, \ldots, B_n\} \subseteq I$, for some $I \in X$
iff $A = solve(\uparrow B) \in T_{PE}(I)$, for some $I \in X$
iff $A \in lub(T_{PE}(X))$

(iii) $A = \downarrow \alpha, solve(\alpha) \leftarrow S_1, \ldots, S_n$ is a ground instance of a clause in P and $\{S_1, \ldots, S_n\} \subseteq lub(X)$
iff $solve(\alpha) \leftarrow S_1, \ldots, S_n$ is a ground instance of a clause in P and $\{S_1, \ldots, S_n\} \subseteq I$, for some $I \in X$
iff $A = \downarrow \alpha \in T_{PE}(I)$, for some $I \in X$
iff $A \in lub(T_{PE}(X))$.■

Proposition 3.3. Let P be a Reflective Prolog definite program and I an extended Herbrand interpretation of P. Then I is a reflective model for P iff $T_{PE}(I) \subseteq I$.

Proof. I is a reflective model for P iff

(i) for each ground instance $A \leftarrow A_1, \ldots, A_n$ of each clause in P, $A \neq solve(\beta)$, $\{A_1, \ldots, A_n\} \subseteq I$ implies:
—$A \subseteq I$, because I is a model of each axiom of P;
—$solve(\uparrow A) \subseteq I$, by the definition of extended Herbrand interpretation;

(ii) for each ground instance $solve(\uparrow A) \leftarrow S_1, \ldots, S_n$ of each solve clause in P, $\{S_1, \ldots, S_n\} \subseteq I$ implies:
—$solve(\uparrow A) \subseteq I$, because I is a model of each axiom of P
—$A \subseteq I$, because I is a model of the reflection axiom $A \leftarrow solve(\uparrow A)$
iff $T_{PE}(I) \subseteq I$.■

Hence the result of van Emden and Kowalski holds (Lloyd 1987, theorem 6.5, p. 38), providing a fixpoint characterization of the least reflective Herbrand model of a Reflective Prolog definite program. The only difference is that we put $T_{PE} \uparrow 0 = \perp$, but this does not affect the proof. The formulation is given below, where RM_P indicates the least reflective Herbrand model of a Reflective Prolog definite program P.

Theorem 3.1. (Fixpoint characterisation of the least reflective Herbrand model) Let P be a Reflective Prolog definite program. Then $RM_P = lfp(T_{PE}) = T_{PE} \uparrow \omega$.■

4. Unification and resolution

4.1. *Unification*

Reflective Prolog's extended resolution includes forms of implicit reflection, which determine the dynamic change of level modelled in the definition of T_{PE}. This requires two kinds of unifications: extended unification, which deals with expressions generated at the same level and is able to unify metalevel terms, in addition to the usual unification of object-level terms; generalized unification, which deals with expressions generated at different levels by referentiation/dereferentiation operations.

The following substitution rules are the basis of both extended and generalized unification, according to the definition of variable assignment given in definition 3.3.

Definition 4.1. A *substitution* θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where $v_1, \ldots, v_n$ are distinct variables, and $\forall\ i \in \{1, \ldots, n\}$, one of the following conditions holds:

(i) v_i is an object variable, and t_i is an object-level term distinct from v_i;
(ii) v_i is a predicate metavariable, and t_i is a predicate name constant;
(iii) v_i is a function metavariable, and t_i is a function name constant;
(iv) v_i is a general metavariable, and t_i is a metalevel term distinct from v_i.■

The definitions of *expression*, *simple expression* (*se*, for short) and *disagreement set* are the usual ones. Given a finite set S of simple expressions, the disagreement set consists as usual of all the subexpressions of each expression in S that do not coincide. We now present the extended unification algorithm, where S denotes a finite set of simple expressions to be unified.

Extended unification algorithm

1. Put $k=0$ and $\sigma_0 = \epsilon$ (empty substitution).
2. If $S\sigma_k$ is a singleton, then stop; σ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$.
3. If there exist v and t in D_k such that one of the following conditions holds:
   (i) t is an object-level term and v is an object variable that does not occur in t;
   (ii) t is a predicate name constant and v is a predicate metavariable;
   (iii) t is a function name constant and v is a function metavariable;
   (iv) t is a metalevel term and v is a general metavariable that does not occur in t

   then put $\sigma_{k+1} = \sigma_k \{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.■

We now introduce the *generalized unification algorithm*. Accordingly, we will speak of *generalized mgu*, or *gmgu*, and we will say that two expressions *g-unify*.

A *generalized simple expression* (*gse*, for short) is either a term, or an atom, or a predicate symbol, or a function symbol. A ground gse is a gse containing no variable symbol as a subexpression.

Generalized unification algorithm

1. Put $k=0$ and $\sigma_0 = \epsilon$ (empty substitution).
2. If $S\sigma_k$ is a singleton, then stop; σ_k is a gmgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$.
3. If there exist a variable v and a gse t in D_k such that one of the following conditions holds:
   (i) v is a predicate metavariable, and t is a predicate symbol;
   (ii) v is a function metavariable, and t is a function symbol;
   (iii) v is a general metavariable, and t is a term not containing v;

   then put $\sigma_{k+1} = \sigma_k \{v/\uparrow t\}$, increment k and go to 2. Otherwise go to 4.
4. If there exist a variable v and a ground gse t in D_k such that one of the

following conditions holds:
(i) v is an object variable, $\downarrow t$ is an object-level term not containing v;
(ii) v is a predicate metavariable, $\downarrow t$ is a predicate name constant;
(iii) v is a function metavariable, $\downarrow t$ is a function name constant;
(iv) v is a general metavariable, $\downarrow t$ is a metalevel term not containing v;
then put $\sigma_{k+1} = \sigma_k \{v/\downarrow t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.■

A gmgu is clearly a substitution, and can be composed with other substitutions (in particular with either mgus or gmgus) in the usual way. Some examples of gmgus are given below, where α is a relation name term and B is an atom.

α = #P($X, "a") B = p(Y,a) (1) gmgu = {#P/⟨p⟩, $X/"Y"}
α = ⟨q⟩({f}("c")) B = q(f(V)) (2) gmgu = {V/c}
α = #R($F("c")) B = t(f(X)) (3) gmgu = {#R/⟨t⟩,$F/{f},X/c}

The gmgu (1) can be obtained as the mgu of α and $\uparrow$B, the gmgu (2) as the mgu of $\downarrow \alpha$ and B, but for gmgu (3) none of these cases holds. In general, if α is a relation name term, B an atom and θ a gmgu of α and B, we have $\alpha\theta = \uparrow(B\theta)$.

For both extended and general unification it is easy to prove (in a way similar to Lloyd 1987, theorem 4.3, p. 25) that, given a finite set S of respectively simple expressions or generalized simple expressions, if S is respectively unifiable or g-unifiable then the unification algorithm terminates and gives respectively an mgu or gmgu for S, otherwise the unification algorithm terminates and reports this fact.

Theorem 4.1. (Reflective Prolog unification theorem) Let S be a finite set of simple expressions. If S is unifiable, then the extended unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the extended unification algorithm terminates and reports this fact. Let S' be a finite set of generalized simple expressions. If S' is g-unifiable, then the generalized unification algorithm terminates and gives a gmgu for S'. If S' is not g-unifiable, then the generalized unification algorithm terminates and reports this fact.■

4.2. *RSLD-resolution*

Derivation by resolution is extended to state the role of base-level clauses as well as meta-evaluation clauses in a proof, where reflection allows the shifting from the base level to the meta-evaluation level and vice versa. The new definition makes use of both unification and generalized unification. The notation is similar to that of Lloyd (1987), with clauses indicated as A←A_1, ..., A_n instead of A:-A_1, ..., A_n. In order to maintain resolution independent of the computation rule, the following safeness condition must hold of goals ref(α,A).

Definition 4.2. A goal ref(α,B) is *safe* if α is a name term and B a term, and one of the following conditions holds:

- both α and B are ground;
- α is a metavariable and B is ground;
- B is a variable and α is ground.■

Some comments are in order about restrictions implied by the above definition. The predefined predicate ref, which declaratively represents the naming relation, is used procedurally for referencing/dereferencing terms. Dereferencing a non-ground name term is not possible: for instance, we cannot know the term whose name is ⟨p⟩($X) if $X is unbound. On the contrary, referencing a non-ground term is possible (since we have names for variable symbols), and is normally done automatically when the level of the computation changes (as shown in the definition of RSLD-resolution). With respect to the explicit use of ref, consider however the goal ←ref(f(X),$Y),p(X). Since X is unbound, we get $Y/{f}("X"). Reversing the order of the subgoals, we might get a different result if p(X) instantiates X. A dynamic check corresponding to definition 4.2 has to be introduced in the language interpreter, since it is not generally possible to foresee whether or not a goal ref(α,A) of an arbitrary program will be safe at run-time.

We can now define the extended resolution principle. Let $\diamond$ be the true clause, i.e. the clause satisfied by every interpretation (for instance p←p is such a clause). Every program can be considered as implicitly including the true clause.

Definition 4.3. (Reflective SLD-resolution, or RSLD-resolution) Let G be a definite goal $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and C a clause. If A_m is the selected atom in G, then G′ is derived from G and C using substitution θ iff one of the following conditions holds:

(i) C is $A \leftarrow B_1, \ldots, B_q$
θ is an mgu of A_m and A
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$

(ii) C is $\text{solve}(\alpha) \leftarrow S_1, \ldots, S_w$,
$A_m \neq \text{solve}(\delta)$
θ is a gmgu of A_m and α
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, S_1, \ldots, S_w, A_{m+1}, \ldots, A_k)\theta$

(iii) A_m is solve(M)
C is $A \leftarrow B_1, \ldots, B_q$
θ is a gmgu of M and A
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$

(iv) A_m is theory_clause(H,B)
C is $A \leftarrow B_1, \ldots, B_q$
θ is both a gmgu of H and A and a gmgu of B and $[B_1, \ldots, B_q]$
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta$

(v) A_m is theory_clause(H,B)
C is $\text{solve}(\alpha) \leftarrow S_1, \ldots, S_w$
$H \neq \uparrow \text{solve}(\delta)$
θ is both an mgu of H and α and a gmgu of B and $[S_1, \ldots, S_w]$
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta$

(vi) A_m is theory_clause($\uparrow$ solve(H),B)
C is $A \leftarrow B_1, \ldots, B_q$
θ is both a gmgu of H and A and a gmgu of B and $[B_1, \ldots, B_q]$
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta$

(vii) A_m is ref(α,B)
C is $\diamond$

θ is a gmgu of α and B
G′ is the goal $\leftarrow(A_1, \ldots, A_{m-1}, A_{m+1}, A_k)\theta$■

RSLD-resolution subsumes SLD-resolution, which corresponds to case (i) above. Cases (ii) and (iii) perform upward and downward reflection respectively. In case (ii), a base-level selected atom is resolved with a meta-evaluation clause, whose conditions appear in the resulting goal G′. Vice versa, in case (iii) a meta-evaluation selected atom is resolved with a base-level clause, whose conditions appear in the resulting goal G′. Cases (iv)–(vi) deal with the distinguished predicate theory_clause, by modelling the behaviour of cases (i)–(iii). Case (vii) takes care of referentiation/dereferentiation.

It is worth noting that, given the selected atom A_m in a definite goal G, and the subsequent goal G′ obtained (as described above) via one resolution step, the newly-selected atom A'_m in G′ is treated in the same way; that is, a proof is in general obtained by an interleaving between base level and meta-evaluation level.

Example 3. Let us reconsider the program of example 1. The steps to prove the goal :-happy(albert) are the following.

G	:-happy(albert)	
C	happy(X):-friend(X,lucy)	(case (i) of definition 4.3)
θ	[X/albert]	
G′	:-friend(albert,lucy)	
$\uparrow A_m$	⟨friend⟩("albert","lucy")	
C	solve(#P($X, $Y)):-symmetric(#P),solve(#P($Y, $X))	(case (ii) of definition 4.3, *shift-up*)
θ	[#P/⟨friend⟩, $X/"albert", $Y/"lucy"]	
G′	:-symmetric(⟨friend⟩),solve(⟨friend⟩("lucy", "albert"))	
C	symmetric(⟨friend⟩)	(case (i) of definition 4.3)
θ	[]	
G′	:-solve(⟨friend⟩("lucy", "albert"))	
C	friend(lucy,albert)	(case (iii) of definition 4.3, *shift-down*)
θ	[]	
G′	□	(empty clause, success)

In summary, a base level goal A can be attempted both via base-level rules such as A:-B_1, ... B_n, and via meta-evaluation rules solve(↑ A):- S_1, ... S_n; conversely, a meta-evaluation level goal solve(↑ A) can be attempted via both meta-evaluation rules and base-level rules.■

The resolution procedure implemented in the interpreter of Reflective Prolog follows a generalized depth-first strategy, where cases (i) and (iii) have higher priority than (ii): each goal is first attempted at the base level and then at the meta-evaluation level. In practice, the interpreter applies the standard Prolog strategy to an internal representation of the program, in which base clauses always precede meta-evaluation clauses. The predefined predicate theory_clause is handled accordingly, with priority to cases (iv) and (vii): a call to theory_clause($A,$B), or to theory_clause(↑ solve($A),$B), with $B unbound, will generate, on backtracking, first the conditions of base-level clauses whose conclusion *g*-unifies with $A, and then the conditions of solve clauses whose

conclusion unifies with solve($A), according in both cases to the order of clauses in the program.

The resolution procedure includes two kinds of checks (implemented in the language interpreter), defined below. Given an object- or metavariable, say X, X *appears* in the term/atom A if either X or one of the corresponding quoted name constants ("X", ""X"", etc.) occurs as a subterm in A. Let a *g*-variant of a goal M be a goal M′ with the same structure (they unify without instantiations), where either the set of variables appearing in M and the set of variables appearing in M′ are disjoint, or every common variable appears in the same position in both M and M′; e.g., p(X, 1, Y) is a *g*-variant of p(A, 1, B) and of p(X, 1, B) but not of p(Y, 1, X). Of course, the only *g*-variant of a ground goal is the goal itself.

(a) Let $G_1, \ldots, G_n$ be the sequence of goals generated in an RSLD-derivation of $P \cup \{\leftarrow G\}$, and let G_i be the first goal of the sequence containing a meta-evaluation atom M=solve(...) as a subgoal. The derivation fails whenever a goal M1=solve(...) appears in G_k for some $k>i$, where M1 is a *g*-variant of M. That is, meta-evaluation of goals which have already been meta-evaluated in a previous step along the same branch of the proof tree is prevented, since it would lead to an infinite loop. This check is sound since it considers goals which are identical, up to variable renaming.

(b) The derivation of $P \cup \{\leftarrow p(\ldots)\}$ fails whenever a subgoal p(..., ⟨p⟩, ...) appears in G_j for some j. This check is sound since, by the definition of a reflective model, no such atom belongs to RM_P.

These checks are not comparable with the more general approaches to loop detection appearing in the literature (among the latest, (Apt *et al.* 1989, Bol 1990)). They only apply to meta-evaluation goals for specific purposes, leaving untouched the loop problem at the base level. Check (b) is aimed at verifying the condition stated syntactically in definition 2.5, and semantically in definition 3.4. Check (a) is aimed at handling those simple loops which arise from a specific conceptual reason: though expressed in clausal form, solve rules are often meant to express equivalences, as in example 1. Check (a) then applies to those solve subgoals to which a rewriting system would apply simplification.

An RSLD-derivation of $P \cup \{\leftarrow M\}$ is *successful* (and is called an *RSLD-refutation*) if it is finite and its last goal is the empty clause. The computed answer θ is the composition (restricted to the variables appearing in M) of the mgus and/or gmgus associated with the n resolution steps. A *failed* RSLD-derivation is a finite derivation that ends in a non-empty goal to which either none of the cases (i)–(vii) of resolution is applicable, or one of checks (a) and (b) is applicable.

Some comments are in order about the syntactic distinction, introduced in section 2, between base level and meta-evaluation level, and the corresponding restriction that forbids the use of meta-evaluation predicates at the base level. This is related to the choice of implicit rather than explicit reflection: in fact, a call to a meta-evaluation predicate at the base level would correspond to a (possibly indirect) call to solve. Such a call is in contrast with the principles of Reflective Prolog (since solve is meant to *declaratively* extend the meaning of

base-level relations) and also unnecessary. Consider for instance a base-level clause such as p(X):- solve(⟨q⟩("a")), which violates definition 2.7: it is easy to see that it is equivalent to p(X):- q(a), since the subgoal q(a) is attempted at both the base level and the meta-evaluation level, by the definition of RSLD-resolution. In conclusion, we believe that implicit reflection encourages a more declarative attitude in using the language, having the same or greater expressivity than explicit reflection and a simpler and more natural semantics. A comparison is not in order here with the uses of explicit reflection for reasoning with multiple theories (e.g. with the predicate demo ("T", "G")), since this paper deals with the definition and use of metaknowledge within a single theory.

The distinction, at the base level, between object level and metalevel (metalevel predicates cannot be used in object-level clauses) is motivated by the point of view that, at each level, the levels below are visible, but not the levels above.

Results about soundness and completeness of RSLD-resolution w.r.t the least reflective Herbrand model of a program are presented in the Appendix.

5. Sample applications

In this section we outline a characterization of one kind of problems for which Reflective Prolog is especially appropriate. As mentioned in the introduction, two kinds of metaprogramming applications can be (roughly) distinguished. A first one has to do with metaprograms in the strict sense, i.e. programs that take other programs as data and either apply transformations to them, or perform some kind of computation according to the structure of proofs (affect unification/control, count inference steps, build the proof tree, etc). A second one deals with representing information on different but related conceptual levels in one and the same program (this is not however to be intended as a sharp distinction, since more experience is needed to discover where the border line, if any, is).

For the first kind of applications, it is still possible in RP to write meta-interpreters which, due to the naming capabilities of RP, enjoy a neat declarative semantics. The basic version is the following:

```
demoRP([]).
demoRP([$A[SB]):-demoRP($A),demoRP($B).
demoRP($A):-theory_clause($A,$B),demoRP($B).
```

For the second kind of applications, no meta-interpreter is needed in RP, due to its reflection capabilities. In these cases, RP allows cleaner and easier definitions, and also avoids (as discussed later) procedural and expressiveness problems that often arise when trying to combine different definitions. It is worth noting that there are applications, apparently of the first kind, which in RP can be treated like those of the second kind. For instance, a 'query the user' capability can be introduced in an RP program (without using a meta-interpreter), by simply adding to it the following meta-evaluation rule, which is automatically applied, on failure, to any goal (by the definition of RSLD-Resolution):

```
solve(#P($A)):-askable(#P),ask(#P($A)).
```

We do not discuss RP meta-interpreters in this paper. Rather, we intend to illustrate the enhanced expressivity, due to reflection, in dealing with those metaprogramming tasks which are not concerned with the proof process in itself. In this section, we discuss two sample kinds of applications of sufficient generality

and complexity to illustrate what programming in RP is like. Other significant applications are discussed in Costantini and Lanzarone (1989, 1992).

5.1. *Calculus of relations*

The theory, or calculus, of relations constitutes a classical part of basic logic (Carnap 1958, Tarski 1965). It involves properties of relations; mainly, classes (unary relations) and binary relations are considered. This subsection begins with the question of transforming classes into binary relations and vice versa, and then it deals with problems of representation and use of properties of binary relations in a logic program. Several sources of limitations that arise in Prolog with respect to these issues are discussed, and it is shown how Reflective Prolog's capability of objectifying relations is suitable to appropriately overcome those limitations.

The ontological assumption of predicate logic is that the world can be described in terms of individuals, classes and relations. Classes, or types, are collections of individuals satisfying one property, and a property is traditionally represented by a unary predicate. When describing a world, one has to decide which entities to represent as individuals, which ones as classes and which ones as relations. For example, the collection of all men may be either represented as a class, like in:

```
man(john).                                                    (5.1)
```

or treated as an individual, like in:

```
is_a(john,man).                                               (5.2)
```

The choice is usually based on the purpose of the description (Kowalski, 1979). Representation (5.2) enables one to consider superclasses, like in:

```
is_a(man,human).                                              (5.3)
is_a(human,animal).
```

and to refer to them by means of variables in rules. For instance, together with (5.2) and (5.3) we may have the rule:

```
is_a(X,Y):-is_a(X,Z),is_a(Z,Y).
```

This expresses the transitivity of the relation is_a, thus allowing inference chains through subclasses and superclasses. (In Prolog, however, this definition involves procedural problems, which will be discussed in the following.) In place of (5.3) we may define the following rules:

```
is_a(X,human):-is_a(X,man).
is_a(X,animal):-is_a(X,human).
```

from which inference chains are automatically obtained, due to the implicit transitivity of the operator :-. Transitivity rules cannot be expressed, and inference chains cannot be obtained, if classes are treated as properties.

Prolog allows a non-uniform representation, where the same symbol is used as a class in one place and as an individual in another, like in:

```
man(john).
human(man).
animal(human).
```

but this is an abuse of logic. It is known (Deliyanni and Kowalski 1979) how to pass from the representation as property to the representation as binary relation (for instance, from (1) to (2)): the unary predicate symbol (man) becomes a constant symbol, and a new binary predicate is introduced (is_a), which expresses how the original predicate symbol (man) is related to its argument (john). However, this transformation, or the choice between the two representations, must be made in advance. It is an *a priori* of a given description, and cannot be modified from the inside of the description itself.

The Reflective Prolog's naming device allows one to reify relations, i.e. to associate metalevel constants with predicate symbols, so that the two representations can coexist without the above-mentioned lack of uniformity, like in:

```
man(john).                                                    (5.4)
is_a(bob,⟨man⟩).
is_a(⟨man⟩,⟨human⟩).
is_a(X,⟨animal⟩):-is_a(X,⟨human⟩).
```

In addition, automatic reflection permits passing from constant symbols to predicate symbols and vice versa. It is, therefore, possible to transform one representation into the other, and to use each of them in the appropriate context. These transformations can be made explicit in the program by means of the following meta-evaluation rules:

```
solve(⟨is_a⟩($X,$Y)):-theory_fact(⟨is_a⟩($Z,$Y)),solve(⟨is_a⟩($X,$Z)).
solve(⟨is_a⟩($X,$Y)):-ref($Y,#P),solve(#P($X)).
solve(#P($X)):-ref($X,Y),is_a(Y,#P).
```

Notice that theory_fact($F) is a specialization of the predefined predicate theory_clause($A,$B) for unit clauses. Here, and in the subsequent examples as well, it can be considered as defined by

```
theory_fact($F):-theory_clause($F,[]).
```

With these, a positive answer is obtained from (5.4) e.g. to the queries

```
?-is_a(john,⟨man⟩).
?-man(bob).
?-is_a(bob,⟨human⟩).
```

by the application of the second, third and first rule, respectively. Note that, since some facts have arguments at different levels, it is necessary to use the ref predicate in the last two solve rules dealing with those cases.

In the following, we will consider binary relations only, recalling Kowalski (1979) and Deliyanni and Kowalski (1979) and that every n-ary relation ($n>2$) can be replaced by $n+1$ binary relations.

The Horn-clause language allows the definition of first-order relationships (relations among individuals) by means of facts, and, partially, of second-order relationships (relations among classes or among relations), by means of the following rules:

```
r2(X,Y):-r1(X,Y). (inclusion: (∀X,Y) X r1 Y ⇒ X r2 Y)
r(X,Y):-r1(X,Y).
r(X,Y):-r2(X,Y). (union:(∀ X,Y) X r Y ⇔ X r1 Y ∨ X r2 Y)
```

```
r(X,Y):-r1(X,Y),r2(X,Y). (intersection: (∀ X,Y) X r Y ⇔ X r1 Y ∧ X r2 Y)
r(X,Y):-r1(X,Z),r2(Z,Y).(composition:(∀ X,Y) ∃ Z   X r Y ⇔ X r1 Z ∧ Z r2 Y)
r2(X,Y):-r1(Y,X). (inverse relation: (∀X,Y) X r1 Y ⇔ Y r2 X)
```

In the above cases, the declarative interpretation of Horn clauses is in agreement (for one direction of the double implication) with the definitions of the theory of relations (Carnap 1958, Tarski 1965), and the procedural interpretation (implemented by the Prolog interpreter) allows the inference of all couples belonging to the relation according to the already known ones (asserted in facts) and to the given rules. Representing general properties of binary relations such as reflexivity, symmetry and transitivity is, however, a different matter. They can be declaratively expressed, but their procedural interpretation is rather troublesome.

In the following, the problems that arise are pointed out, and different possible solutions are discussed. The properties of a relation r are meant as defined over the class C of the individuals appearing in facts concerning that relation, according to the definitions below.

```
r reflexive in C:      (∀ X ∈ C) X r Y ⇒ X r X ∧ Y r Y
r symmetric in C:      (∀ X,Y ∈ C) X r Y ⇒ Y r X
r transitive in C:     (∀ X,Y,Z ∈ C) X r Y ∧ Y r Z ⇒ X r Z.
```

Reflexivity in a class can be expressed neither in the simple form

```
r(X,X).
```

since this would apply to all the individuals of the Herbrand Universe, nor in the declaratively correct form

```
r(X,X):-r(X,_).
r(X,X):-r(_,X).
```

since procedurally this definition could lead to infinite loops, e.g. with:

```
r(a,b).
?-r(b,b).
```

It is necessary to introduce an auxiliary relation:

```
rr(X,X):-r(X,_).
rr(X,X):-r(_,X).
```

Similarly, symmetry cannot be expressed by the rule:

```
r(X,Y):-r(Y,X).
```

since this can determine infinite loops, for instance when arguments are instantiated to constants that do not belong to the class. Again, an auxiliary relation is needed:

```
rr(X,Y):-r(X,Y).
rr(X,Y):-r(Y,X).
```

The direct representation of transitivity:

```
r(X,Y):-r(X,Z),r(Z,Y).
```

causes infinite loops with (partially) unistantiated calls. Transitivity has be be expressed as:

```
rr(X,Y):-r(X,Y).
rr(X,Y):-r(X,Z),rr(Z,Y).
```

Thus, for each reflexive or symmetric or transitive relation, an auxiliary relation must be introduced, which obscures the declarativeness of the representation. Moreover, for those transitive relations which can naturally be expressed as transitive closures of subrelations, it is not always possible to represent all the desired connections. Given, for instance, the following clauses and query:

```
parent(a,b).
parent(c,d).
ancestor(b,c).
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
?-ancestor(X,Y).
```

the solutions (a,d) and (b,d) are not obtained. To yield them, the clause:

```
ancestor(X,Y):-parent(Z,Y),ancestor(X,Z).
```

(which generates repetitions) must be added. But in this case, by also adding the fact:

```
ancestor(c,e).
```

the query above fails to generate the couples (a,e) and (b,e).

These problems become even more serious when properties of relations are considered not in isolation, as before, but combined. In fact, given the clauses:

```
p(X,Y):-p(Y,X).
p(X,Y):-p(X,Z),p(Z,Y).
p(a,b).
p(c,b).
```

the Prolog interpreter enters an infinite loop with the query:

```
?-p(a,c).
```

though the program is declaratively correct and p(a,c) is a logic consequence of it. It is well-known (Lloyd 1987) that no clause reorder and no predefined selection and search rules can solve this problem. It is also worth noting that the same problems occur not only with Prolog, but with other representation and inference systems too (Shapiro 1986).

In Reflective Prolog, we can specify which relation has which property by means of assertions like:

```
reflexive(⟨r1⟩).
symmetric(⟨r2⟩).
transitive(⟨r3⟩).
```

and then use the following meta-evaluation rules.

For reflexivity:

```
solve(#P($X,$X)):-reflexive(#P),solve_1(#P,$X).
solve_1(#P,$X):-theory_fact(#P($X,_)).
solve_1(#P,$X):-theory_fact(#P(_,$X)).
```

For symmetry:

```
solve(#P($X,$Y)):-symmetric(#P),solve(#P($Y,$X)).                    (5.5)
```

For direct transitivity:

```
solve(#P($X,$Y)):-transitive(#P),theory_fact(#P($X,$Z)), solve(#P($Z,$Y)).
                                                                     (5.6)
```

For transitivity of a relation r expressed as transitive closure of a subrelation r1, we first assert:

```
subsumes(⟨r⟩,⟨r1⟩).
```

and then use the following rules:

```
solve(#P($X,$Y)):-fact(#P($X,$Y)).
solve(#P($X,$Y)):-transitive(#P),fact(#P($X,$Z)),solve(#P($Z,$Y)).
fact(#P($X,$Y)):-theory_fact(#P($X,$Y)).                             (5.7)
fact(#P($X,$Y)):-subsumes(#P,#Q),theory_fact(#Q($X,$Y)).
```

In order to combine symmetry and direct transitivity, it suffices to replace (or to add before) clause (5.7) with the clause:

```
fact(#P($X,$Y)):-symmetric(#P),theory_fact(#P($Y,$X)).
```

Now we call attention to the consequences of the already-mentioned fact that some second-order relationships can be expressed at the object level in one direction only ('if', not 'iff'). The inverse relation cannot be expressed in both directions:

```
r2(X,Y):-r1(Y,X).
r1(X,Y):-r2(Y,X).
```

since this would lead to infinite loops. In Reflective Prolog there are two possibilities. The first one is to represent the 'if' part at the base level, and the 'only if' part at the meta-evaluation level:

```
r2(X,Y):-r1(Y,X).
inverse(⟨r1⟩,⟨r2⟩).                                                  (5.8)
solve(#P($X,$Y)):-inverse(#P,#Q),solve(#Q($Y,$X)).                   (5.9)
```

The second possibility is to have clause (5.8) together with the following:

```
symmetric(⟨inverse⟩).
```

and to use rule (5.9) together with the symmetry rule (5.5).

In addition to representing second-order relationships (more extensively than in Prolog), Reflective Prolog also allows the representation of definitions and theorems of relational calculus. Similarly to inverse relation, inclusion cannot be expressed at the object level in both directions. Thus, the identity rule (if r1 $\subset$ r2 and r2 $\subset$ r1, then r1 = r2) cannot be represented. In RP, mutual inclusion can be specified by two assertions:

```
includes(⟨1⟩,⟨r2⟩).
includes(⟨r2⟩,⟨r1⟩).                                    (5.10)
```

and identity can be represented by the meta-evaluation rule:

```
solve(⟨identity⟩(#P,#Q)):-includes(#P,#Q),includes(#Q,#P).    (5.11)
```

An alternative solution is to directly specify the identity of two relations:

```
identity(⟨r1⟩,⟨r2⟩).                                    (5.12)
symmetric(⟨identity⟩).
```

and then to use these assertions to assign the known extension of one of the two relations to the other one by means of the rule:

```
solve(#P($X,$Y)):-identity(#P,#Q),solve(#Q($Y,$Y)).
```

Note that the last rule is usable both when the identity of two relations is directly asserted with clauses (5.12), and when it is derivable via rule (5.11) from assertions (5.10). We now proceed to consider other metatheorems.

Every symmetric and transitive relation is also reflexive (on its domain):

```
reflexive(#P):-symmetric(#P), transitive(#P).
```

If a relation is symmetric, then its inverse relation is also symmetric:

```
symmetric(#P):-inverse(#P,#Q),symmetric(#Q).             (5.13)
```

More generally, if a relation has one property, then its inverse has the same property. Metarules similar to (5.13) could be added, for example for reflexive and transitive. But the metatheorem can be expressed at its proper level of generality by means of the following metarule only:

```
property(#P,#Q):-inverse(#P,#R),property(#R,#Q).         (5.14)
```

Rule (5.14) assumes that a property is represented in the form:

```
property(⟨rel⟩,⟨prop⟩).
```

rather than:

```
property(⟨rel⟩).
```

However, according to the results of the beginning of this Subsection, it is possible to pass from one form to the other by means of the following meta-evaluation rules:

```
solve(#P($X)):-ref($X,#Q),property(#Q,#P).               (5.15)
solve(⟨property⟩($X,$Y)):-ref($Y,#P),solve(#P($X)).      (5.16)
```

Given, for example:

```
symmetric(⟨r1⟩).
property(⟨r1⟩,⟨reflexive⟩).
inverse(⟨r2⟩,⟨r1⟩).
```

by using (5.14), (5.15) and (5.16) the following can be derived:

```
symmetric(⟨r2⟩).
property(⟨r2⟩,⟨symmetric⟩).
```

```
reflexive(⟨r2⟩).
property(⟨r2⟩,⟨reflexive⟩).
```

Another important metatheoretical notion is the following (Carnap 1958, p. 146). A property p is hereditary with respect to a relation r (or p is preserved under r) if, whenever a member of r has property p, then so do all the other members of r:

$$\text{p hereditary w.r.t. r: } (\forall \text{ X, Y} \in \text{C})\ \text{pX} \wedge \text{X r Y} \Rightarrow \text{pY} \tag{5.17}$$

For each relation r which preserves some property p (and for each property p preserved by r), we can express the Prolog rule:

```
p(Y):-r(X,Y),p(X).                                    (5.18)
```

or else, representing a property with a binary predicate:

```
p(Y,Value):-r(X,Y),p(X,Value).                        (5.19)
p(Y,Value):-r(Y,X),p(X,Value).
```

In RP we can declaratively assert which relation preserves which property:

```
hereditary(⟨r⟩,⟨p⟩).
```

and define hereditary by either the rule:

```
solve(#P($Y)):-hereditary(#R,#P),solve(#R($X,$Y)),solve(#P($X)).   (5.20)
```

or the rule:

```
solve(#P($Y,$V)):-hereditary(#R,#P),
solve(#R($Y,$X)),solve(#P($X,$V)).                    (5.21)
```

The RP representation (5.21) has several advantages over the Prolog representation (5.19), that we show in the following comparison between Prolog and Reflective Prolog solutions.

One advantage is that several object-level rules of type (5.19) are condensed into one, both for different predicates and the same relation (first and second case below), and for different relations and the same predicate (second and third case below):

```
owner(Y,V):-part_of(Y,X),owner(X,V).
owner(car,o).
part_of(battery,car).
?-owner(battery,V).          /* V = o */
in (Y,V):-part_of(Y,X),in(X,V).
in(car,location).
part_of(battery,car).
?-in(battery,V).             /* V = location */

in(Y,V):-on(Y,X),in(X,V).
in(truck,location).
on(box,truck).
?-in(box,V). /* V = location */
```

Given, together with the same facts as above:

```
hereditary(⟨part_of⟩,⟨owner⟩).
hereditary(⟨part_of⟩,⟨in⟩).
hereditary(⟨on⟩,⟨in⟩).
```

by using rule (5.21) the same answers to the previous queries are obtained. The second advantage is that rule (5.21), having two solve calls in its body, can be combined with any other meta-evaluation rule. For instance, it allows transitivity of both the property and the relation:

```
in(truck,manhattan).
on(box,truck).
on(dog,box).
part_of(manhattan,new_york_N_Y).
part_of(new_york_N_Y,new_york).
transitive(⟨on⟩).
transitive(⟨part_of⟩).
```

By using, together with rule (5.21), the solve rule (5.6) for direct transitivity, the following query is answered affirmatively:

```
?-in(dog,new_york).
```

As an additional advantage, an 'upward' heredity, in place of the previous 'downward' heredity, can be expressed by means of inverse relations:

```
hereditary(⟨contains⟩,⟨defective⟩).
inverse(⟨contains⟩,⟨part_of⟩).
defective(battery).
part_of(battery,car).
```

By using rule (5.20) in combination with rule (5.9), the following query gets a positive answer:

```
?-defective(car).
```

Finally, rules like (5.18) can be extended to second-order relationships. For example:

```
symmetric(⟨friend⟩).
translation(⟨amico⟩,⟨friend⟩).
hereditary(⟨translation⟩,⟨symmetric⟩).
```

It may be noted that (5.21) generalizes the most common forms of inheritance used in semantic nets. Let is_a(X,Y) mean that element X belongs to set (or type, or kind) Y, and a_k_o(X,Y) (short for 'a kind of') that set X is a subset of Y. Then, by instantiating #R to ⟨is_a⟩ or ⟨a_k_o⟩ in (5.21), we have

```
solve(#P($Y,$V)):-hereditary(⟨is_a⟩,#P),
                  solve(⟨is_a⟩($Y,$X)),solve(#P($X,$V)).        (5.22)
solve(#P($Y,$V)):-hereditary(⟨a_k_o⟩,#P),
                  solve(⟨a_k_o⟩($Y,$X)),solve(#P($X,$V)).       (5.23)
```

which express the inheritance laws of is_a and a_k_o (thus giving the semantics of these predicates):

if X has relation P to V and Y is an X, then Y has relation P to V (5.24)

if X has relation P to V and Y is a kind of X, then Y has relation P to V (5.25)

In semantic-net systems, these rules have to be either represented as auxiliary nets, or implemented in the net interpreter. Note that rules (5.22) and (5.23) require, to be applied, the presence of assertions of the form

```
hereditary(⟨is_a⟩,#P).
hereditary(⟨a_k_o⟩,#P).
```

Such assertions enable the differentiation of a distributive property (a property that is true of the members of a group) and a collective property (a property that is true of a group itself, not its members). Properties of the first kind, but not of the second kind, are to be asserted as hereditary. Given, for instance

```
origin_period(mammals,triassic).
a_k_o(equines,mammals).
temperament(domestic_animal,docile).
a_k_o(cat,domestic_animal).
hereditary(⟨a_k_o⟩,⟨temperament⟩).
```

from rules (5.22) and (5.23) temperament(cat,docile) can be correctly derived, since this is a distributive and therefore hereditary property. Instead, origin_period(equines,triassic) (false in the intended interpretation) cannot and must not be derived, since origin_period is a collective property and therefore has not been asserted as hereditary. (The terms 'collective' and 'distributive' properties and the last example are taken from Richards (1989).)

5.2. *Equality by function names*

In this subsection we show some examples about using names of function symbols and metavariables ranging over them, and discuss how they can approximate a kind of equality in Horn clauses.

First, the following simple example illustrates the use of the three types of names and metavariables:

```
divertente(spettacolo(burattini)).                                    (5.26)
translation(⟨amusing⟩,⟨divertente⟩).
translation({performance},{spettacolo}).
translation("puppets","burattini").
solve(#P($X)):-translation(#P,#Q),solve(#(Q($X)).
solve(#P(%F($X))):-translation(%F,%G),solve(#P(%G($X))).
solve(#P(%F($X))):-translation($X,$Y),solve(#P(%F($Y))).
```

From fact (5.26), all the eight ground atoms can be derived, which are obtained by representing either in Italian or in English any of the three (predicate, function and constant) symbols.

It is worth noting that in RP a query can be expressed at any level. With respect to the above example, the following (meta) queries:

```
?-translation($X,$Y).
?-solve(#P(%F($X))).
```

obtain three and eight (meta) solutions, respectively.

We will now reconsider, and extend, the representation of equality of some

geometric figures, discussed in Kornfeld (1983). We summarize the problem as follows:

(i) there is a class of objects, regular polygons, which consists of various subclasses (equilateral triangles, squares, pentagons, etc.);
(ii) the subclasses are characterized by an attribute (side length), and the class by two attributes (number of sides, side length);
(iii) the elements of the class, and thus of the subclasses, have geometric features, like area and perimeter, represented by binary predicates having a (sub)class as the first argument and a value as the second argument;
(iv) the values of those geometric features are computable from the values of the mentioned attributes, with the known geometric rules;
(v) each subclass has a proper name which can be considered as synonymous with the name of the class, qualified by the corresponding number of sides; e.g., square can be considered as synonym of regular_polygon (4);
(vi) we wish to use the same geometric rules given for the class to compute the geometric features of any subclass, identified by its proper name.

The rules mentioned in point (iv) can be represented at the object level:

```
perimeter(regular_polygon(Sides_Number,Side_Length),Perimeter:-
                    Perimeter is Sides_Number * Side_Length.
area(regular_polygon(Sides_Number,Side_Length),Area):-
                    L_Square is Side_Length *Side_Length,
                    Cot is cotangent(180/Sides_Number),
                    Area is (L_Square * Sides_Number * Cot) / 4.
```

The synonymy features mentioned in point (v) can be represented by the meta-assertions:

```
equivalent({equilateral_triangle}($L),{regular_polygon}("3",$L)).
equivalent({square}($L),{regular_polygon}("4",$L)).
equivalent({pentagon}($L),{regular_polygon}("5",$L)).
.....
```

or, still better, by parametrizing the equivalent assertion and separately representing the association between the proper name of a subclass and the number of sides of the corresponding regular polygon:

```
equivalent(%F($L),{regular_polygon}($N,$L)):-number_of_sides(%F,$N).
number_of_sides({equilateral triangle},"3"). number_of_sides({square},"4").
number_of_sides({pentagon},"5").
.....
```

Given the clauses above, the objective mentioned in point (vi) is reached by means of the simple meta-evaluation clause:

```
solve(#P($X,$Y)):-equivalent($X,$Z),solve(#P($Z,$Y)).          (5.27)
```

We have, for instance

```
?-perimeter(square(10),P).                    /* P = 40 */
?-area(equilateral_triangle(100),A).          /* A= 4330 */
```

In a similar way, we can represent other classes and subclasses of geometric figures:

```
area(ellipse(Major_Axis,Minor_Axis),Area):-
        Area is 3.14 * (Major_Axis / 2) * (Minor_Axis / 2).
times(X,Y,Z):-Z is X * Y.
equivalent({circle}($Radius),{ellipse}($Diameter,$Diameter)):-
        solve(⟨times⟩("2",$Radius,$Diameter)).
```

By means of the same rule (5.27), a query about the area of a circle is solved by using the rule for the area of an ellipse:

```
?-area(circle(10),A). /* A = 314 */
```

On the other hand, as also was the case in Kornfeld (1983), the area rule for ellipses will not interfere in any way with goals asking for the areas of regular polygons, because there is no assertion that allows us to derive equivalence between ellipses and regular polygons.

It may be noted that rule (5.27) expresses the important higher-order logic postulate that 'identical objects share the same properties' (Rogers 1971).

Kornfeld (1983) has discussed how his Prolog-with-equality (inclusion of assertions about equality, used to prove two terms equal when their unification is attempted and fails syntactically) is a natural extension to the power, expressibility and generality of standard Prolog. Equality theorems corresponding to those expressible in Kornfeld's Prolog-with-equality are in RP just a special case of solve rules. In this regard, RP is not only more powerful, but also has procedural and logical semantics, lacking in Prolog-with-equality.

6. Discussion

We have already argued that Prolog's metalinguistic features are unsuitable as a full naming device, since they do not allow both a clear distinction and an adequate communication among different representation levels. We have also mentioned that a part of the logic programming community has critically analysed the language in this regard, and is actively working toward its evolution. In this section we compare Reflective Prolog with the well-known meta-interpreters approach. Then we discuss the matter of efficiency of RP, also with respect to meta-interpreters.

6.1. *Comparison with Prolog meta-interpreters*

In order to examine more closely the Prolog limitations mentioned above, let us reconsider example 1 of section 2. Since properties of relations (symmetry, equivalence) are involved, a representation at the object level would suffer from the shortcomings discussed in the previous section. A general solution can only be expressed at the metalevel, by defining a meta-interpreter, like the following:

```
demo(true):-!.
demo((A,B)):-!,demo(A),demo(B).
demo(A):-clause(A,B),demo(B).
demo(A):-A=..[P,X,Y],symmetric(P),B=..[P,Y,X],demo(B).
```

```
demo(A):-A=..[P,X.Y],equivalent(P,P1),B=..[P1,X,Y],demo(B).
```

to be used, in the example, with the clauses

```
friend(giorgio,mary).
symmetric(friend).
equivalent(amico,friend).
symmetric(equivalent).
amico(lucy,albert).
happy(X):-friend(X,lucy).
```

The queries have now to be expressed through the predicate demo

```
?-demo(amico(mary,giorgio)).                                    (6.1)
```

This solution is inefficient, since all subgoals are meta-interpreted, not only those that need to be so treated. This is better seen by adding the following clauses:

```
play_tennis(X,Y):-plays_tennis(X),plays_tennis(Y),amico(X,Y).
plays_tennis(giorgio).
plays_tennis(mary).
```

The query

```
?-demo(play_tennis(mary,giorgio)).
```

leads to meta-interpret not only the third subgoal, but also the first two. On the other hand, if the alternate solution

```
play_tennis(X,Y):-plays_tennis(X),plays_tennis(Y),demo(amico(X,Y)).
?-play_tennis(mary,giorgio).
```

is adopted, one has to decide in advance which goals are to be meta-interpreted and which are not, the former needing to be included in predicate demo.

Another problem is the difficulty in avoiding the possibility of infinite loops, which arises when the basic ('vanilla') interpreter is augmented with several additional clauses. In the example, the additional clauses are those for symmetry and equivalence, and an infinite loop arises e.g. with the queries

```
?-demo(friend(charles,anna)).                                   (6.2)
?-demo(amico(charles,anna)).                                    (6.3)
```

because the symmetry clause is applied indefinitely. This difficulty can be overcome neither by adding cuts in the meta-interpreter's code, e.g. in the clause before the last:

```
demo(A):-A=..[P,X,Y],symmetric(P),!,B=..[P,Y,X],demo(B).
```

nor by pushing the execution of the new subgoals to the object level, i.e. by replacing the last two clauses with the following:

```
demo(A):-A=..[P,X,Y],symmetric(P),B=..[P,Y,X],B.
demo(A):-A=..[P,X,Y],equivalent(P,P1),B=..[P1,X,Y],B.
```

These techniques inhibit the combined application of properties. For instance, the query (6.1) cannot be proved. In fact, once the equivalence clause is applied, the goal friend(mary,giorgio) fails: to succeed, the application of the symmetry

clause would be needed, but is precluded. In order to solve the problem, it is necessary to extend the meta-interpreter with control features, so that it records which clauses have already been applied and avoids using them over and over:

```
demo(true):-retract(used(_)),fail.
demo(true):-!.
demo((A,B)):-!,demo(A),demo(B).
demo(A):-clause(A,B),demo(B).
demo(A):-A=..[P,X,Y],symmetric(P),not used(symmetric(P)),
        assert(used(symmetric(P))),B=..[P,Y,X],demo(B).
demo(A):-A=..[P,X,Y],equivalent(P,P1),not used(equivalent(P,P1)),
        assert(used(equivalent(P,P1))),B=..[P1,X,Y],demo(B).
```

This second version of the meta-interpreter rules out the occurrence of infinite loops, such as those arising with queries (6.2) and (6.3), but while the first version was already procedural in nature, this one is almost unreadable. Still more important is that it gives an incorrect (negative) answer to queries like:

```
?-demo(happy(albert)).                                    (6.4)
```

since it is unable to apply the assertion symmetric(equivalent) in order to derive the conclusion equivalent(friend,amico) that is needed for the proof. The problem is that, while in the RP solution the two rules for symmetry and equivalence are expressed independently of each other, and they only interact at run-time, this does not occur with the above meta-interpreter. An alternative possibility is that of combining symmetry and equivalence into additional meta-interpreter rules. With respect to coping with the mentioned loop problems, instead of using assert and retract, another way is that of introducing an additional parameter in the meta-interpreter, representing the list of the already meta-interpreted goals, and forcing to failure the meta-interpretation of a goal belonging to the list. The resulting meta-interpreter (Yalcinap 1991) is the following:

```
demo(true,_S):-!.
demo((A,B),S):-!,demo(A,S),demo(B,S).
demo(Goal,S):-clause(Goal,Body),demo(Body,S).
demo(Goal,S):-symmetric_goals(Goal,New_Goal),
        not in_stack(Goal,S),
        demo(New_Goal,[Goal|S]).
demo(Goal,S):-equiv_goals(Goal,New_Goal),
        not in_stack(Goal,S),
        demo(New_Goal[Goal|S]).
equiv_goals(G1,G2):-G1=..[Name|Args],
        equivalent(Name,Name2),
        G2=..[Name2|Args].
equiv_goals(G1,G2):-G1=..[Name2|Args],
        equivalent(Name,Name2),
        G2=..[Name|Args].
symmetric_goals(G1,G2):-G1=..[Name,Arg1,Arg2],
        symmetric(Name),
        G2=..[Name,Arg2,Arg1].
in_stack(Goal,[GoalB|S]):-variant(Goal,GoalB),!.
```

```
in_stack(Goal,[_|S]):-in_stack(Goal,S).
```

(with variant(G1,G2) suitably defined).

This meta-interpreter is more readable than the previous one, and gives the correct positive answer to query (6.4). However, it does so not by using the assertion symmetric(equivalent) (that could therefore be dropped from the database), but because of the two clauses for equiv_goals, which try an equivalent assertion in both directions. The knowledge about symmetry and equivalence properties of relations has thus been turned from declarative to procedural form. Moreover, this formulation is not only longer, but also less general than the one in RP. Given, for instance

```
equivalent(symmetric,invertible).
invertible(r).
r(a,b).
```

the meta-interpreter fails to positively answer the query:

```
?-demo(r(b,a)).                                    (6.5)
```

because the subgoals symmetric(r), equivalent(r,Name2), equivalent(Name,r) all fail. To accommodate this case, another, perhaps inevitably again *ad hoc*, reformulation of the meta-interpreter would be needed. It seems that there is no way of writing a meta-interpreter capable of treating in full generality the combination of properties accomplished by the RP solution. The reason is that, while each subgoal is attempted in all possible ways by the RP's reflection mechanism before failing definitively, it is abandoned by the meta-interpreter on the first failure to proceed to other clauses. In the last example, RP correctly handles query (6.5) because, when the subgoal symmetric(⟨r⟩) fails at the object level, the subgoal equivalent(symmetric,#Q) is generated, which instantiates #Q to ⟨invertible⟩, so that invertible(⟨r⟩) can be used.

In general, Reflective Prolog programs are easier to develop than the corresponding Prolog meta-interpreters for at least two reasons. First, the aspects related to the proof process need not be made explicit in those metaprogramming applications which are not concerned with them. Second, it is not necessary to cope explicitly with procedural problems related to the combination of rules, which are automatically dealt with at the interpreter level.

Explicit loop controls are not needed in a Reflective Prolog program. The interpreter of RP automatically inhibits the repeated meta-evaluation of the same goal, by means of the checks(a) and (b) illustrated in section 4. In the above example, it proves all of the queries by never reapplying the same property to the same subgoal.

We now briefly summarize the main aspects illustrated by the previous examples. First, Reflective Prolog has a general naming device, whereby lower-level terms are represented by higher-level names, which are terms themselves. This feature overcomes the syntactical limitations of Prolog, and makes its metalogical predicates unnecessary. Semantic ambiguities are avoided by having different representations for constant, function and predicate symbols, and the only way to pass from one representation to the other (e.g. from an object-level predicate symbol to a metalevel name constant) is by means of the reference/dereference operation, which is automatically performed by the extended resolution procedure.

The role that the various levels of knowledge play in a demonstration is determined by a general and powerful inference device, which uses reflection to switch the computation context betwen the base level and the meta-evaluation level. This mechanism cannot be simulated by a Prolog meta-interpreter. Prolog allows two rudimentary forms of reflection, schematizable as follows:

```
demo(A):-A.                                    (6.6)
B:-demo(B).                                    (6.7)
```

Rule (33) pushes to the object level the execution of A and of all the subgoals generated by resolution during the proof of A. Further meta-interpretation is, therefore, impossible, unless it is planned in advance for specific subgoals by means of rules like (6.7). On the contrary, rule (6.7) forces meta-interpretation of all the subgoals of B, except those excluded in advance by means of rules like (6.6). In Reflective Prolog, a continuous interleaving between base level and meta-evaluation level occurs automatically, with no need for planning ahead in the program: every subgoal generated by a resolution step is attempted at the base level first, and only if this fails, it is meta-evaluated.

6.2. *Remarks on efficiency*

As discussed in the previous sections, reflection allows metalevel knowledge to be expressed directly, without defining a meta-interpreter, and thus without the extra level of interpretation. In Reflective Prolog, non-failing computations, being dealt with directly by the interpreter, can in principle be as efficient as in Prolog. The situation is, however, different in case of failures. Consider an RP program with typical solve clauses like those for symmetry and transitivity, shown in section 5, and with the following usual definition of membership:

```
member(E,[E|_T]).
member(E,[_H|T]):-member(E,T).
```

With the query ?-member(X,[a,b,c]), X is quickly bound to a, b and c. Now consider the subsequent failure: the interpreter will try to determine if member is symmetric, transitive, etc., for any property of binary predicates the meta-evaluation level knows about. Presumably, all these will fail, but meta-evaluation will be very costly even for such a simple object-level operation.

Explicit reflection, allowing to plan ahead a shift among levels, might in some cases reduce inefficiency, but in our opinion is not a suitable general solution, because of the reasons discussed in section 6.1. What we think can really help are program analysis and transformation techniques.

We have defined a program analyser and transformer for RP programs (Costantini *et al.* 1991), which performs two tasks. First, it identifies those predicates to which no meta-evaluation rule is applicable (i.e. all of them would fail); for these predicates, it generates a predefined directive which is accepted by the interpreter, and prevents their meta-evaluation. This directive would of course be potentially unsafe if issued by the user in an uncontrolled way. Second, it identifies the subset of meta-evaluation rules applicable to each of the other predicates, and (on request) specializes solve rules accordingly.

We have proved the correctness and completeness of this analyser for 'pure' RP programs (i.e. programs containing no extra-logical features), in the sense that it never rules out, for a predicate, meta-evaluation clauses that might be

applicable to it, while it always succeeds in detecting that a given predicate cannot be meta-evaluated at all, or that it cannot be meta-evaluated by a certain rule. For programs with extra-logical predicates, however, the analyser is not complete, though covering most cases in practice. The analyser has been written in RP itself, also in order to show the applicability of RP to this class of metaprogramming applications. It is based on an elaboration of the basic RP meta-interpreter shown at the beginning of section 5.

7. Conclusions and future directions

This paper has presented Reflective Prolog, a new logic programming language that addresses the problem of representing knowledge and metaknowledge in a uniform, declarative way. It avoids the creation of a superior/inferior relationship between levels, which, on the contrary, are defined so as to be able to interleave and cooperate. Reflective Prolog extends the Horn-clause language with an appropriate naming relation and a form of logic reflection. The extension is syntactically minimal and natural, and has semantically all the classical properties. The treatment of reflection with comparatively few semantics extensions is one result of this research, since up to now reflection has been considered as a feature desirable in principle, but semantically troublesome.

In effect, one main effort in developing the language has been that of not departing significantly from the conventional declarative and procedural semantics of logic programming languages (Lloyd 1987). One basic reason for this is to be able to inherit those theories of program transformation, partial evaluation, abstract interpretation, etc., that have been developed on that basis. Not considering the naming features (and the related extensions to unification), since a naming device is necessary in every language which allows declarative metaprogramming, the main differences are those aimed at modelling reflection, i.e. the following. For declarative semantics, we consider a subset of the set of conventional Herbrand interpretations (thus a restriction rather than a modification); we employ the usual notion of model and logical consequence on an extended version of the program (augmented by the reflection axioms). This means that we do not modify the logic, but rather the program. For procedural semantics, resolution has (aside from the predefined predicates) three cases instead of one. The proofs appearing in this paper show that, with respect to the conventional semantics, some of them remain unchanged, and others need only minor technical modifications, within the same conceptual framework.

In parallel with the theoretical work, we have also carried on the implementation of the language interpreter. A first prototype was written in Prolog and enabled us to refine the language definition through experimentation of its use. A second implementation, in C language, was made by modifying the public-domain C-Prolog interpreter. Both interpreters have been equipped with debugging facilities.

The work presented in this paper is part of an ongoing wider project, proceeding along the following lines.

- Self-reference and reflection are introduced into a Horn-clause language as a first step, which is the content of the present paper. Names are given to terms and predicates only. A class of interesting problems can be coped with this 'core' Reflective Prolog, as exemplified in section 5.
- As a second step, this language is extended with a 'metalevel negation'. This

means the declarative definition of what is intended not to hold in a theory and the inferential use of such information to restrict the reflective models of the theory. Metalevel negation does not rely on a negation operator, but rather on negative metarules, which allow the definition of negative knowledge and the drawing of negative conclusions. By means of these rules, either operators (such as negation as failure) or other ways of treating negative information can be defined. Negative metarules are integrated in the inference process by means of reflection, in the style of RP: positive and negative information can therefore fully interact with each other. Semantics of Reflective Prolog with metalevel negation is based on the kind of treatment of logic reflection presented in this paper. Several problems of non-monotonic reasoning can be represented with such augmented Reflective Prolog. First results on this topic are reported in Costantini and Lanzarone (1990).

- Self-reference is then extended to clauses, so that several reasoning agents can be represented, by means of multiple theories. This requires constructs for the representation and manipulation of object-level clauses at the metalevel, more sophisticated than the simple one (theory_clause) that was used in the present paper. Primitives for communication among theories are also defined, again having logical reflection as the underlying procedural and semantic principle. An initial definition of communicating agents in RP is reported in Costantini *et al.* (1992), where problems like the three wise men puzzle are solved using these primitives.
- A further stage is to endow the language with the capability of representing and reasoning with propositional attitudes, such as knowledge and belief. The idea is to build this capability on top of the already existing features of the language, in the line of syntactic treatments of modalities. This work is currently just beginning.

We intend to pursue these aims with regard to all linguistic, semantic, implementation and application aspects, as we have done up to now. We have started to study the applicability of Reflective Prolog to the treatment of incomplete knowledge and of assumption-based reasoning, like defaults, non-monotonic reasoning (Costantini and Lanzarone 1990), some forms of inheritance and of analogical reasoning (Costantini and Lanzarone 1992). The long-term goal of this research is, in fact, to enhance logic programming capabilities in order to deal with complex and sophisticated problems, like those encountered in the development of knowledge-based systems.

Acknowledgements

We gratefully acknowledge interactions with J. Barklund, G. Degli Antoni, M. Gelfond, K. Konolige, R. Kowalski, G. Levi, J. Lloyd, D. Miller, L. Sterling, V. S. Subrahmanian and U. Yalcinap, which stimulated us to clarify several points of this work. Any remaining misconceptions are of course our entire responsibility.

References

Abramson, H. D. and Rogers, M. H. (eds) (1989) Meta-Programming in logic programming. *Proceedings of the First International Workshop* Bristol, UK, 22–24 June, 1988 (Cambridge, Mass: MIT Press).

Aiello, L., Cecchi, C. and Sartini, D. (1986) Representation and use of metaknowledge. *Proceedings of the IEEE,* **74**(10): 1304–1321.

Apt, K. R., Bol, R. N. and Klop, J. W. (1991) An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, **86**(1): 35–60.

Benjamin, P. (1990) A metalevel manifesto. Brazdil, P. B. and Konolige, K. (eds) *Machine Learning, Meta-reasoning and Logics* (Boston: Kluwer Academic Publishers) pp. 3–17.

Bol, R. N. (1990) Towards more efficient loop checks. Debray, S. and Hermenegildo, M. (eds) *Logic Programming. Proceedings of the 1990 North-American Conference* (Cambridge, Mass: MIT Press) pp. 465–479.

Bowen, K. A. and Kowalski, R. A. (1982) Amalgamating language and metalanguage. Clark, K. L. and Tarnlund, S.-A. (eds), *Logic Programming.* (London: Academic Press) 153–172.

Bowen, K. A. and Weinberg, T. (1985) A meta-level extension of Prolog. Cohen, J. and Conery, J. (eds) *Proceedings of the 1985 Symposium on Logic Programming* (Washington, D.C.: IEEE Computer Society Press) pp. 48–53.

Bruynooghe, M. (ed.) (1990) *Proceedings of the 2nd Workshop on Metaprogramming in Logic (Meta90)*, Leuven, Belgium, 4–6 April 1990.

Carnap, R. (1958) *Introduction to Symbolic Logic and its Applications* (New York: Dover).

Chen, W., Kifer, M. and Warren, D. S. (1989) HiLog: a first-order semantics for higher-order logic programming constructs. *Proceedings of the 1989 North-American Conference on Logic Programming*, Cleveland, Ohio, 16–20 October, 1989.

Costantini, S., Dell'Acqua, P. and Lanzarone, G. A. (1992) Reflective agents in metalogic programming. A. Pettorossi (ed.) *Proceedings of the Third International Workshop on Metaprogramming in Logic (Meta92).* In *Lecture Notes in Artificial Intelligence*. p. 649 (Berlin: Springer-Verlag).

Costantini, S. and Lanzarone, G. A. (1988) Towards metalogic programming. Martelli, A. and Valle, G. (eds) *Computational Intelligence I. Proceedings of Computational Intelligence '88* (Amsterdam: North-Holland) pp. 41–52.

Costantini, S. and Lanzarone, G. A. (1989) A metalogic programming language. Levi, G. and Martelli, M. (eds), *Logic Programming. Proceedings of the Sixth International Conference* (Cambridge, Mass: MIT Press) pp. 218–233.

Costantini, S. and Lanzarone, G. A. (1990) Metalevel negation and non-monotonic reasoning. To appear in *Methods of Logic in Computer Science*. Abstract in *Proceedings of the Workshop on Logic Programming and Non-Monotonic Reasoning*, Austin, TX, 1–2 November 1990; pp. 19–26.

Costantini, S. and Lanzarone, G. A. (1992) Analogical reasoning in reflective Prolog. Martino, A. (ed.) *Expert Systems in Law* (Amsterdam: Elsevier Science Publisher) pp. 133–145.

Costantini, S., Lanzarone, G. A., Laudani, A. and Zanzi, A. (1991) On the optimization of metalogic programs. Asirelli, P. (ed.) *Proceedings of the Sixth Italian Conference on Logic Programming*, Pisa, June 1991, pp. 259–273.

Davies, N. (1990) Towards a first order theory of reasoning agents. Carlucci Aiello, L. (ed.) *Proceedings of the 9th European Conference on Artificial Intelligence* (London: Pitman) pp. 195–200.

Deliyanni, A. and Kowalski, R. A. (1979) Logic and semantic networks. *Communications of the ACM,* **22**(3): 184–192.

Feferman, S. (1972) Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic,* **27**: 259–316.

Hill, P. M. and Lloyd, J. W. (1988) Analysis of meta-programs. Abramson, H. D. and Rogers, M. H. (eds) *Meta-programming in Logic Programming* (Cambridge, Mass: MIT Press) pp. 23–32.

Hill, P. M. and Lloyd, J. W. (1991) The Godel Report. TR-91-02. Department of Computer Science, University of Bristol.

Kowalski, R. A. (1979) *Logic for Problem Solving* (Amsterdam: North-Holland Elsevier).

Kornfeld, W. (1983) Equality for Prolog. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence I* (Palo Alto, CA: Morgan Kaufmann) pp. 514–519.

Lloyd, J. W. (1987) *Foundations of Logic Programming*, second, extended edition (Berlin: Springer-Verlag).

Miller, D. and Nadathur, G. (1988) An overview of λ-Prolog. Bowen, K. A. and Kowalski, R. A. (eds) *Logic Programming. Proceedings of the Third International Conference and Symposium I* (Cambridge, Mass: MIT Press) pp. 810–827.

Perlis, D. (1985) Languages with self-reference I: foundations. *Artificial Intelligence*, **25**: 301–322.

Perlis, D. (1988) Meta in Logic. Maes, P. and Nardi, D. (eds) *Meta-level Architectures and Reflection* (Amsterdam: North-Holland) pp. 37–49.

Pylyshym, Z. W. (1984) *Computation and Cognition* (Cambridge, Mass: MIT Press).

Richards, T. (1989) *Clausal Form Logic—An Introduction to the Logic of Computer Reasoning.* (Sydney: Addison-Wesley).

Rogers, R. (1971) *Mathematical Logic and Formalized Theories* (Amsterdam: North-Holland).
Shapiro, S. C. (1986) Symmetric Relations, Intensional Individuals, and Variable Binding. *Proceedings of the IEEE,* **74**(10): 1354–1363.
Smith, B. C. (1984) Reflection and Semantics in Lisp. Xerox Parc ISL-5. Palo Alto (CA).
Sterling, L. and Shapiro, E. (1986) *The Art of PROLOG—Advanced Programming Techniques* (Cambridge, Mass: MIT Press).
Subrahmanian, V. S. (1988) A simple formulation of the theory of metalogic programming. Abramson, H. D. and Rogers, M. H. (eds) *Meta-programming in Logic Programming* (Cambridge, Mass: MIT Press) pp. 65–80.
Tarnlund, S.-A. (1977) Horn clause computability. *BIT,* **17**(2): 215–226.
Tarski, A. (1965) *Introduction to Logic, and to the Methodology of Deductive Sciences* 3rd edition (Oxford: Oxford University Press).
van Harmelen, F. (1989) A classification of meta-level architectures. Jackson, P., Reichgelt, H. and van Harmelen, F. (eds) *Logic-based Knowledge Representation* (Cambridge: MIT Press) pp. 13–35.
Weyhrauch, R. W. (1980) Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence,* 13: 133–170.
Yalcinap, L. U. (1991) Meta-Programming for Knowledge Based Systems in Prolog. PhD thesis TR 91–141. Department of Computer Engineering and Science, Case Western Reserve University.

Appendix

We present here the results of soundness and completeness of RSLD-resolution w.r.t. the least reflective Herbrand model of a program. In the following, let P be a Reflective Prolog definite program, RM_P the least reflective Herbrand model of P and G a definite goal. The proofs resemble those for classical SLD-resolution, which can be regarded as a particular case of RSLD-resolution where definition 4.3 is restricted to point (i).

The definitions of *RSLD-refutation, success set, answer, computed answer* are exactly as usual (Lloyd 1987), where 'definite program' must be intended as 'Reflective Prolog definite program', 'SLD-resolution' as 'RSLD-resolution' and 'mgu' as 'mgu or gmgu'. We report below the definition of computed answer for the sake of clarity.

Definition A.1 Let P be a Reflective Prolog definite program and G a definite goal. A computed answer for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1, \ldots, \theta_n$ to the variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of mgus and/or gmgus used in an RSLD-refutation of $P \cup \{G\}$.■

The definition of correct answer differs from the usual one, since it deals with reflective logical consequence.

Definition A.2. Let P be a Reflective Prolog definite program, G a definite goal $\leftarrow \mathsf{A}_1, \ldots, \mathsf{A}_k$ and θ an answer for $P \cup \{G\}$. We say that θ is a correct answer for $P \cup \{G\}$ if $\forall(\mathsf{A}_1 \wedge \ldots \wedge \mathsf{A}_k)\theta)$ is a reflective logical consequence of P.■

As usual, we say that the answer 'no' is correct if $P \cup \{G\}$ is satisfiable. It is straightforward to prove (as in Lloyd 1987, Theorem 6.6, p. 40) that an answer θ is correct iff $(\mathsf{A}_1, \ldots, \mathsf{A}_k)\theta$ is true w.r.t. every reflective Herbrand model of P, iff $(\mathsf{A}_1, \ldots, \mathsf{A}_k)\theta$ is true w.r.t. the least reflective Herbrand model of P.

In the following, we say that an atom A is *non-distinguished* if $\mathsf{A}=\mathsf{p}(\ldots)$, $\mathsf{p} \neq \mathsf{solve} \wedge \mathsf{p} \neq \mathsf{theory\_clause} \wedge \mathsf{p} \neq \mathsf{ref}$. We abbreviate reflective logical consequence with rlc. It is also useful to recall that a reflective Herbrand model (definition 3.5) is, in particular, an extended Herbrand interpretation (definition 3.4).

A.1. Soundness

Theorem A.1 (Soundness of RSLD-resolution). Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

Proof. Let G be the goal $\leftarrow A_1, \ldots, A_k$ and $\theta_1, \ldots, \theta_n$ the sequence of mgus and/or gmgus used in a refutation of $P \cup \{G\}$. We prove that $\forall((A_1, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P by induction on the length of the refutation.

Suppose first that $n = 1$. This means that G is a goal of the form $\leftarrow A_1$, and one of the following cases holds.

(i) P has a unit clause of the form $A\leftarrow$ and $A_1\theta_1 = A\theta_1$. Since $A_1\theta_1$ is an instance of a unit clause in P, it follows that $\forall(A_1\theta_1)$ is a rlc of P.

(ii) A_1 is non-distinguished; P has a unit clause of the form $\mathsf{solve}(\alpha)\leftarrow$ and $\uparrow(A_1\theta_1) = \alpha\theta_1$. Since $\mathsf{solve}(\uparrow(A_1\theta_1))$ is an instance of a unit clause in P, it follows that $\forall(\mathsf{solve}(\uparrow(A_1\theta_1)))$ is a logical consequence of P, and then, by the reflection axioms, $\forall(A_1\theta_1)$ is a rlc of P.

(iii) $A_1 = \mathsf{solve}(M)$; P has a unit clause of the form $A\leftarrow$ and $\uparrow(A\theta_1) = M\theta_1$. Since $A\leftarrow$ is a program clause, $\forall(A)$ is a rlc of P and so is $\forall(A\theta_1)$; then, by the definition of reflective Herbrand model (which is in particular an extended Herbrand interpretation), $\forall(\mathsf{solve}(\uparrow(A\theta_1)))$ is also a rlc of P.

(iv) $A_1 = \mathsf{theory\_clause}(M,B)$; P has a clause of the form $A\leftarrow B_1, \ldots, B_n$ and $\uparrow(A\theta_1) = M\theta_1$, $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)] = B\theta_1$. Since $A\leftarrow B_1, \ldots, B_n$ is a program clause, by the definition of reflective Herbrand model $\forall(\mathsf{theory\_clause}(\uparrow(A\theta_1),[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]))$ is a rlc of P.

(v) $A_1 = \mathsf{theory\_clause}(M,B)$; P has a clause of the form $\mathsf{solve}(A)\leftarrow B_1, \ldots, B_n$ and $A\theta_1 = M\theta_1$, $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)] = B\theta_1$. Since $\mathsf{solve}(A)\leftarrow B_1, \ldots, B_n$ is a program clause, by the definition of reflective Herbrand model $\forall(\mathsf{theory\_clause}((A\theta_1),[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]))$ is a rlc of P.

(vi) $A_1 = \mathsf{theory\_clause}(M1,B)$, $M1 = \uparrow\mathsf{solve}(M)$; P has a clause of the form $A\leftarrow B_1, \ldots, B_n$ and $\uparrow(A\theta_1) = M\theta_1$, $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)] = B\theta_1$. Since $A\leftarrow B_1, \ldots, B_n$ is a program clause, by the definition of reflective Herbrand model $\forall(\mathsf{theory\_clause}(\uparrow(\mathsf{solve}(\uparrow(A\theta_1))),[\uparrow B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]))$ is a rlc of P.

(vii) $A_1 = \mathsf{ref}(\alpha,A)$, A_1 is safe, and $\uparrow(A\theta_1) = \alpha\theta_1$. By the definition of reflective Herbrand model, $\forall(\mathsf{ref}(\alpha,A)\theta_1)$, which is the same as $\mathsf{ref}(\alpha,A)\theta_1$, is a rlc of P.

Next, suppose that the result holds for computed answers coming from refutations of length $n-1$, and that $\theta_1, \ldots, \theta_n$ is the sequence of mgus and/or gmgus used in a refutation of length n. Let C be the first input clause and A_m the selected atom of G. One of the following cases holds.

(i) C is $A\leftarrow B_1, \ldots, B_q (q \geq 0)$.
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. Thus, if $q > 0$, $\forall((B_1, \ldots, B_q)\theta_1 \ldots \theta_n)$ is a rlc of P. Consequently, $\forall((A_m)\theta_1 \ldots \theta_n)$, which is the same as $\forall((A)\theta_1 \ldots \theta_n)$, is a rlc of P. Hence $\forall((A_1, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P.

(ii) C is solve(α)←S_1, ..., S_w ($w \geq 0$), A_m is non-distinguished.
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, S_1, \ldots, S_w, A_{m+1}, \ldots, A_k)$ is a rlc of P. Thus, if $w > 0$, $\forall((S_1, \ldots, S_w)\theta_1 \ldots \theta_n)$ is a rlc of P. Consequently, $\forall((\text{solve}(\alpha))\theta_1 \ldots \theta_n)$, which is the same as $\forall(\text{solve}(\uparrow((A_m)\theta_1 \ldots \theta_n)))$, is a rlc of P, and, by the reflection axioms, $\forall((A_m)\theta_1 \ldots \theta_n)$ is a rlc of P. Hence $\forall((A_1, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P.

(iii) C is A←B_1, ..., B_q($q \geq 0$). A_m is solve(M).
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. Thus, if $q > 0$, $\forall((B_1, \ldots, B_q)\theta_1 \ldots \theta_n)$ is a rlc of P. Consequently, $\forall((A)\theta_1 \ldots \theta_n)$ is a rlc of P. Then, by the definition of reflective Herbrand model, $\forall((A_m)\theta_1 \ldots \theta_n)$, which is the same as $\forall(\text{solve}(\uparrow((A)\theta_1 \ldots \theta_n)))$, is a rlc of P. Hence $\forall((A_1, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P.

(iv) C is A←B_1, ..., B_q, A_m is theory_clause(M,B).
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. By the definition of reflective Herbrand model, $\forall((A_m)\theta_1 \ldots \theta_n)$, which is the same as $\forall(\text{theory\_clause}(\uparrow((A)\theta_1 \ldots \theta_n), [\uparrow(B_1)\theta_1 \ldots \theta_n), \ldots, \uparrow((B_n(\theta_1 \ldots \theta_n)]))$ is a rlc of P.

(v) C is solve(A)←B_1, ..., B_q, A_m is theory_clause(M,B).
By the induction hypothesis, $\forall((A_1, \ldots, A_{-1}, A_{m+1}, \ldots A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. By the definition of reflective Herbrand model, $\forall((A_m)\theta_1 \ldots \theta_n)$, which is the same as $\forall(\text{theory\_clause}(((A)\theta_1 \ldots \theta_n), [\uparrow((B_1)\theta_1 \ldots \theta_n), \ldots, \uparrow((B_n)\theta_1 \ldots \theta_n)]))$ is a rlc of P.

(vi) C is A←B_1, ... B_q, A_m is theory_clause(M1,B), M1 = $\uparrow$(solve(M)).
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, A_{m+1}, \ldots A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. By the definition of reflective Herbrand model, $\forall((A_m)\theta_1 \ldots \theta_n)$, which is the same as $\forall(\text{theory\_clause}(\uparrow(\text{solve}(\uparrow((A)\theta_1 \ldots \theta_n))), [\uparrow((B_1)\theta_1 \ldots \theta_n), \ldots, \uparrow((B_n)\theta_1 \ldots \theta_n)]))$ is a rlc of P.

(vii) C is $\Diamond$, A_m is ref(α,A), A_m is safe, and $\uparrow(A\theta_1 \ldots \theta_n) = \alpha\theta_1 \ldots \theta_n$.
By the induction hypothesis, $\forall((A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P. By the definition of reflective Herbrand model, $\forall((\text{ref}(\alpha,A)\theta_1 \ldots \theta_n))$, which is the same as $\text{ref}(\alpha,A)\theta_1 \ldots \theta_n$, is a rlc of P. Hence $\forall((A_1, \ldots, A_k)\theta_1 \ldots \theta_n)$ is a rlc of P.■

Corollary A.1. Suppose there exists an RSLD-refutation of $P \cup \{G\}$. Then $P \cup \{G\}$ is unsatisfiable.■

Corollary A.2. The success set of Prolog is contained in RMP.■

Now we strengthen corollary A.2, showing that if $A \in B_{PE}$ and $P \cup \{\leftarrow A\}$ has an RSLD-refutation of length n, then $A \in T_{PE} \uparrow n$. This is an extension of the result due to Apt and van Emden, reported in Lloyd (1987, theorem 7.4, p. 44).

If A is an atom, we put $[A] = \{A' \in B_{PE} : A' = A\theta$, for some substitution $\theta\}$. Thus [A] is the set of all ground instances of A.

Theorem A.2. Let G be a definite goal $\leftarrow A_1 \ldots, A_k$. Suppose that $P \cup \{G\}$ has an RSLD-refutation of length n and that $\theta_1, \ldots, \theta_n$ is the sequence of mgus and/or gmgus of the RSLD-refutation. Then

$$\cup_{j=1}^{k} [A_j\theta_1 \ldots \theta_n] \subseteq T_{PE} \uparrow n.$$

Proof. The result is proved by induction on the length of the refutation. Suppose first that $n = 1$. Then G is a goal of the form $\leftarrow A_1$, and one of the following cases holds.

(i) P has a unit clause of the form $A\leftarrow$ and $A_1\theta_1 = A\theta_1$. Clearly, $[A] \subseteq \text{TPE} \uparrow 1$, and so does $[A_1\theta_1]$.

(ii) A_1 is non-distinguished; P has a unit clause of the form $\text{solve}(\alpha)\leftarrow$ and $\uparrow(A_1\theta_1) = \alpha\theta_1$. Clearly, $[\text{solve}(\uparrow(A_1\theta_1))] \subseteq \text{TPE} \uparrow 1$, and so does $[A_1\theta_1]$, by the definition of TPE.

(iii) $A_1 = \text{solve}(M)$; the program has a unit clause of the form $A\leftarrow$ and $\uparrow(A\theta_1) = M\theta_1$. Clearly, $[A_1\theta_1] \subseteq \text{TPE} \uparrow 1$, and so does $[\text{solve}(M)]$, which is the same as $[\text{solve}(\uparrow(A_1\theta_1))]$, by the definition of TPE.

(iv) $A_1 = \text{theory\_clause}(M,B)$; P has a clause of the form $A\leftarrow B_1, \ldots, B_n$ and $\uparrow(A_1\theta_1) = M\theta_1$, $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)] = B\theta_1$. Then, $[\text{theory\_clause}(\uparrow(A\theta_1),[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])) \subseteq \text{TPE} \uparrow 0$ by the definition of TPE, and $\text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow 1$ by the monotonicity of TPE.

(v) $A_1 = \text{theory\_clause}(M,B)$; P has a clause of the form $\text{solve}(A)\leftarrow B_1, \ldots, B_n$ and $A\theta_1 = M\theta_1$, $[B_1\theta_1, \ldots, B_n\theta_1] = B\theta_1$. Then, $[\text{theory\_clause}((A)\theta_1,[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0$ by the definition of TPE, and $\text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow 1$ by the monotonicity of TPE.

(vi) $A_1 = \text{theory\_clause}(M1,B)$, $M1 = \uparrow \text{solve}(M)$; P has a clause of the form $A\leftarrow B_1, \ldots, B_n$ and $\uparrow(A_1\theta_1) = M\theta_1$, $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)] = B\theta_1$. Then, $[\text{theory\_clause}(\uparrow(\text{solve}(\uparrow(A\theta_1))),[\uparrow B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0$ by the definition of TPE, and $\text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow 1$ by the monotonicity of TPE.

(vii) $A_1 = \text{ref}(\alpha,A)$, A_1 is safe, and $(A\theta_1) = \alpha\theta_1$. Then, $[\text{ref}(\alpha,A)\theta_1] \subseteq \text{TPE}0$ by the definition of TPE, and $\text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow 1$ by the monotonicity of TPE.

Next, suppose that the result holds for refutations of length $n-1$ and consider a refutation of length n. Let A_j be an atom of G. Suppose first that A_j is not the selected atom of G. Then $A_j\theta_1$ is an atom of G_1, the second goal of the refutation. The induction hypothesis implies that $[A_j\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow (n-1)$ and $\text{TPE} \uparrow (n-1) \subseteq \text{TPE} \uparrow (n)$, by the monotonicity of TPE. Now suppose that A_j is the selected atom of G. Let C be the first input clause. One of the following cases holds.

(i) C is $A\leftarrow B_1, \ldots, B_q (q \geq 0)$.
Then $A_j\theta_1$ is an instance of A. If $q = 0$, we have $[A] \subseteq \text{TPE} \uparrow 1$. Thus $[A_j\theta_1 \ldots \theta_n] \subseteq [A_j\theta_1] \subseteq [A] \subseteq \text{TPE} \uparrow 1 \subseteq \text{TPE} \uparrow n$. If $q > 0$, by the induction hypothesis, $[B_i\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n-1$, for $i=1, \ldots, q$. By the definition of TPE, we have $[A\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n$, and so does $[A_j\theta_1 \ldots \theta_n]$.

(ii) C is $\text{solve}(\alpha)\leftarrow S_1, \ldots, S_w (w \geq 0)$, A_j is non-distinguished.
Then $\uparrow(A_j\theta_1)$ is an instance of α. If $w = 0$, we have $[\text{solve}(\alpha)] \subseteq \text{TPE} \uparrow 1$, which is the same as $[\text{solve}(\uparrow(A_j\theta_1))] \subseteq \text{TPE} \uparrow 1$. Thus by the definition of TPE, we have $[A_j\theta_1] \subseteq \text{TPE} \uparrow 1$. Then, $[A_j\theta_1 \ldots \theta_n] \subseteq [A_j\theta_1] \subseteq \text{TPE} \uparrow 1$. If $w > 0$, by the induction hypothesis, $[S_i\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n-1$, for $i=1, \ldots, w$. By the definition of TPE, we have $[\text{solve}(\alpha)\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n$, which is the same as $[\text{solve}(\uparrow(A_j\theta_1 \ldots \theta_n))] \subseteq \text{TPE} \uparrow n$. thus, again by the definition of TPE, $[A_j\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n$.

(iii) C is $A \leftarrow B_1, \ldots, B_q (q \geq 0)$. A_j is solve(M).
Then $M\theta_1$ is an instance of $\uparrow(A\theta_1)$. If $q = 0$, we have $[A\theta_1] \subseteq [A] \subseteq \text{TPE} \uparrow 1$. Thus by the definition of TPE, we have $[\text{solve}(\uparrow A\theta_1)] \subseteq \text{TPE} \uparrow 1$. Then, $[\text{solve}(M)\theta_1 \ldots \theta_n] = [\text{solve}(\uparrow((A)\theta_1 \ldots \theta_n))] \subseteq [\text{solve}(\uparrow(A\theta_1))] \subseteq \text{TPE} \uparrow 1 \subseteq \text{TPE} \uparrow n$. If $q > 0$, by the induction hypothesis $[B_i\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n-1$, for $i=1, \ldots, q$. By the definition of TPE, we have $[A\theta_1 \ldots \theta_n] \subseteq \text{TPE} \uparrow n$. Again by the definition of TPE, we have $[\text{solve}(M)\theta_1 \ldots \theta_n] = [\text{solve}(\uparrow((A)\theta_1 \ldots \theta_n)] \subseteq \text{TPE} \uparrow n$.

(iv) C is $A \leftarrow B_1, \ldots, B_n$, A_j is theory_clause(M,B).
Then $M\theta_1$ is an instance of $\uparrow(A\theta_1)$, and $B\theta_1$ is an instance of $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]$. by the definition of TPE, we have $[\text{theory\_clause}(\uparrow(A\theta_1), [\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0$. Then, $[\text{theory\_clause}((M)\theta_1 \ldots \theta_n, (B)\theta_1 \ldots \theta_n)] = [\text{theory\_clause}(\uparrow(A\theta_1 \ldots \theta_n), [\uparrow(B_1\theta_1 \ldots \theta_n), \ldots, \uparrow(B_n\theta_1 \ldots \theta_n)])] \subseteq [\text{theory\_clause}(\uparrow(A\theta_1), [\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow n$.

(v) C is $\text{solve}(A) \leftarrow B_1, \ldots, B_n$, A_j is theory_clause(M,B).
Then $M\theta_1$ is an instance of $A\theta_1$, and $B\theta_1$ is an instance of $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]$. By the definition of TPE, we have $[\text{theory\_clause}((A\theta_1), [(B_1\theta_1), \ldots, (B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0$. Then, $[\text{theory\_clause}((M\theta_1 \ldots \theta_n), (B\theta_1 \ldots \theta_n)] = [\text{theory\_clause}((A\theta_1 \ldots \theta_n), [\uparrow(B_1\theta_1 \ldots \theta_n), \ldots, \uparrow(B_n\theta_1 \ldots \theta_n)])] \subseteq [\text{theory\_clause}((A\theta_1), [\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow n$.

(vi) C is $A \leftarrow B_1, \ldots, B_n$, A_j is theory_clause(M1, B), $M1 = \uparrow(\text{solve}(M))$.
Then $M\theta_1$ is an instance of $\uparrow(A\theta_1)$, and $B\theta_1$ is an instance of $[\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)]$. By the definition of TPE, we have $[\text{theory\_clause}(\uparrow \text{solve}(\uparrow(A\theta_1))), [\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0$. Then, $[\text{theory\_clause}((M1\theta_1 \ldots \theta_n), (B\theta_1 \ldots \theta_n)] = [\text{theory\_clause}(\uparrow(\text{solve}(\uparrow(A\theta_1 \ldots \theta_n)), [\uparrow(B_1\theta_1 \ldots \theta_n), \ldots, \uparrow(B_n\theta_1 \ldots \theta_n)])] \subseteq [\text{theory\_clause}(\uparrow(\text{solve}(\uparrow(A\theta_1))), [\uparrow(B_1\theta_1), \ldots, \uparrow(B_n\theta_1)])] \subseteq \text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow n$.

(vii) C is $\diamond$, A_m is $\text{ref}(\alpha, A)$, A_m is safe, and $\uparrow(A\theta_1 \ldots \theta_n) = \alpha\theta_1 \ldots \theta_n$.
Then $\alpha\theta_1$ is an instance of $\uparrow(A\theta_1)$. By the definition of TPE, we have $[(\text{ref}(\alpha, A)\theta_1)] \subseteq \text{TPE} \uparrow 0$. Then, $[(\text{ref}(\alpha, A)\theta_1 \ldots \theta_n)] \subseteq [(\text{ref}(\alpha, A)\theta_1)] \subseteq \text{TPE} \uparrow 0 \subseteq \text{TPE} \uparrow n$.■

A.2. Completeness

An *unrestricted* RSLD-refutation is an RSLD-refutation, except that the substitutions θ_i are no longer required to be most general unifiers or g-unifiers, but are only required to be unifiers or g-unifiers.

Lemma A.1. (Extended Mgu lemma) Suppose that $P \cup \{G\}$ has an unrestricted RSLD-refutation. Then $P \cup \{G\}$ has an RSLD-refutation of the same length with mgus and/or gmgus $\theta'_1, \ldots, \theta'_n$ such that, if $\theta_1 \ldots, \theta_n$ are the unifiers and/or g-unifiers from the restricted RSLD-refutation, then there exists a substitution γ such that $\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\gamma$.

Proof. The result is proved by induction on the length of the unrestricted refutation. Suppose first that $n = 1$. Thus $P \cup \{G\}$ has an unrestricted RSLD-refutation $G_0 = G$, $G_1 = \square$ with input clause C_1 and unifier or g-unifier θ_1. One of the following cases holds.

(i) θ_1 is a unifier of the atom in G and the head of the unit clause C_1;
(ii) C_1 is solve(α)←, θ_1 g-unifies A and α, where A is the atom in G;
(iii) C_1 is A←, the atom in G is solve(M), θ_1 g-unifies M and A;
(iv) C_1 is A←B_1, ..., B_n, the atom in G is theory_clause(M,B), θ_1 g-unifies M and A as well as B and [B_1, ..., B_n];
(v) C_1 is solve(A)←B_1, ..., B_n, the atom in G is theory_clause(M,B), θ_1 unifies M and A and g-unifies B and [B_1, ..., B_n];
(vi) C_1 is A←B_1, ..., B_n, the atom in G is theory_clause(M1,B), M1 = ↑solve(M), θ_1 g-unifies M and A as well as B and [B_1, ..., B_n];
(vii) C_1 is ◇, the atom in G is ref(α,A), θ_1 g-unifies α and A.

Assume respectively that:

(i) θ'_1 is an mgu of the atom in G and the head of the unit clause C_1;
(ii) θ'_1 is a gmgu of A and α;
(iii) θ'_1 is a gmgu of M and A;
(iv) θ'_1 is a gmgu of M and A as well as of B and [B_1, ..., B_n];
(v) θ'_1 is an mgu of M and A as well a gmgu of B and [B_1, ..., B_n];
(vi) θ'_1 is a gmgu of M and A as well as of B and [B_1, ..., B_n];
(vii) θ'_1 is a gmgu of α and A.

Then $\theta_1 = \theta'_1\gamma$, for some γ, and $P \cup \{G\}$ has an RSLD-refutation $G_0 = G$, $G_1 = \square$ with input clause C_1 and mgu or gmgu θ_1.

Now suppose that the result holds for $n-1$. Suppose that $P \cup \{G\}$ has an unrestricted RSLD-refutation $G_0 = G$, G_1, ..., $G_n = \square$ with input clauses C_1, ..., C_n and unifiers and/or g-unifiers θ_1, ..., θ_n. There exists one of the following:

(i) an mgu θ'_1 for the selected atom in G and the head of C_1
(ii) a gmgu θ'_1 for A and α, where A is the selected atom in G, and solve(α) is the head of C_1
(iii) a gmgu θ'_1 for M and A, where solve(M) is the selected atom in G, and A is the head of C_1
(iv) a gmgu θ'_1 for M and A as well as for B and [B_1, ..., B_n], where theory_clause(M,B) is the selected atom in G, and A←B_1, ..., B_n is the clause C_1
(v) an mgu θ'_1 for M and A which is also a gmgu for B and [B_1, ..., B_n], where theory_clause(M,B) is the selected atom in G, and solve(A)←B_1, ..., B_n is the clause C_1
(vi) a gmgu θ'_1, for M and A as well as for B and [B_1, ..., B_n], where theory_clause(M1,B) is the selected atom in G, M1 = ↑solve(M), and A←B_1, ..., B_n is the clause C_1.
(vii) a gmgu θ'_1 for α and A, where ref(α,A) is the selected atom in G, and C_1 is ◇

such that $\theta_1 = \theta'_1\rho$, for some ρ. Thus $P \cup \{G\}$ has an unrestricted RSLD-refutation $G_0 = G$, G'_1, ..., $G_n = \square$ with input clauses C_1, ..., C_n and unifiers and/or g-unifiers θ'_1, $\rho\theta_2$, ..., θ_n, where $G_1 = G'_1\rho$. By the induction hypothesis, $P \cup \{G'_1\}$ has an RSLD-refutation $G_0 = G$, G'_1, G'_2, ..., $G'_n = \square$ with mgus and/or gmgus θ'_2, ..., θ'_n such that $\rho\theta_2 \ldots \theta_n = \theta'_2 \ldots \theta'_n\gamma$, for some γ. Thus $P \cup \{G\}$

has an RSLD-refutation $G_0 = G, G_1, \ldots G_n = \square$ with mgus and/or gmgus θ'_1, ..., θ'_n such that $\theta_1 \ldots \theta_n = \theta'_1\rho\theta_2 \ldots \theta_n = \theta'_n\gamma$.■

Lemma A.2. (Extended lifting lemma) Let G be a definite goal and θ a substitution. Suppose there exists an RSLD-refutation of $P \cup \{G\theta\}$. Then there exists an RSLD-refutation of $P \cup \{G\}$ of the same length such that, if θ_1, ..., θ_n are the mgus and/or gmgus from the RSLD-refutation of $P \cup \{G\theta\}$, and θ'_1, ..., θ'_n are the mgus and/or gmgus from the RSLD-refutation of $P \cup \{G\}$, then there exists a substitution γ such that $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\gamma$.

Proof. Suppose that the first input clause for the refutation of $P \cup \{G\theta\}$ is C_1, that the first mgu or gmgu is θ_1 and that the goal resulting from the first step is G_1. We can assume that θ acts on no variable of C_1. Now $\theta\theta_1$ is a unifier or g-unifier for one of the following:

(i) the atom in G which corresponds to the selected atom in $G\theta$ and the head of C_1;
(ii) A and α, where A is the atom in G which corresponds to the selected atom in $G\theta$, and solve(α) the head of C_1;
(iii) M and A, where solve(M) is the atom in G which corresponds to the selected atom in $G\theta$, and A the head of C_1;
(iv) M and A as well as B and $[B_1, \ldots, B_n]$, where theory_clause(M,B) is the atom in G which corresponds to the selected atom in $G\theta$, and A←B_1, ..., B_n is the clause C_1;
(v) M and A as well as B and $[B_1, \ldots, B_n]$, where theory_clause(M,B) is the atom in G which corresponds to the selected atom in $G\theta$, and solve(A)←B_1, ..., B_n is the clause C_1;
(vi) M and A as well as B and $[B_1, \ldots, B_n]$, where theory_clause(M1,B) is the atom in G which corresponds to the selected atom in $G\theta$, M1 = ↑ solve(M), and A←B_1, ..., B_n is the clause C_1;
(vii) α and A, where ref(A,α) is the atom in G which corresponds to the selected atom in $G\theta$, and C_1 is $\diamond$.

The result of resolving G and C_1 using $\theta\theta_1$ is exactly G_1. Thus we obtain an unrestricted RSLD-refutation of $P \cup \{G\}$, which looks exactly like the given RSLD-refutation of $P \cup \{G\theta\}$, except that the original goal is different and the first unifier or g-unifier is $\theta\theta_1$. Now we apply the extended mgu lemma.■

The *success set* of a Reflective Prolog definite program P is defined, as usual, as the set of all A ∈ Bpe such that $P \cup \{\leftarrow A\}$ has an RSLD-refutation. The theorem below extends the result due to Apt and van Emden, reported in [Lloyd 1987, theorem 8.3, p. 48].

Theorem A.3. The success of P is equal to RMp.

Proof. By corollary A.2, it suffices to show that RMp is contained in the success set of P. Suppose that A is in RMp. By theorem A.2 above, A ∈ Tpe ↑ n, for some $n \in \omega$. We prove by induction on n that A ∈ Tpe ↑ n implies that $P \cup \{\leftarrow A\}$ has a refutation and hence A is in the success set.

Suppose first that n=1. Then A ∈ Tpe ↑ 1 means one of the following.

(i) A is a ground instance of a unit clause in P. Then, by the definition of RSLD-resolution (point (i)), $P \cup \{\leftarrow A\}$ has an RSLD-refutation.

(ii) A is non-distinguished; solve(α) is a ground instance of a unit clause in P, and $\alpha = \uparrow$A. Then, by the definition of RSLD-resolution (point (ii)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

(iii) A=solve(α); B is a ground instance of a unit clause in P, and $\uparrow$B = α. Then, by the definition of RSLD-resolution (point (iii)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

(iv) A = theory_clause(α,β); T$\leftarrow$B$_1$, ..., B$_n$ is a ground instance of a clause in P, and $\uparrow$T = α, [$\uparrow$B$_1$, ..., $\uparrow$B$_n$] = β. Then, by the definition of RSLD-resolution (point (iv)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

(v) A = theory_clause(α,β); solve(T)$\leftarrow$B$_1$, ..., B$_n$ is a ground instance of a clause in P, and T = α, [$\uparrow$B$_1$, ..., $\uparrow$B$_n$] = β. Then, by the definition of RSLD-resolution (point (v)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

(vi) A = theory_clause(α,β); T$\leftarrow$B$_1$, ..., B$_n$ is a ground instance of a clause in P, and $\uparrow$(solve($\uparrow$T)) = α, [$\uparrow$B$_1$, ..., $\uparrow$B$_n$] = β. Then, by the definition of RSLD-resolution (point (vi)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

(vii) A = ref(α,A); $\uparrow$A = α. Then, by the definition of RSLD-resolution (point (vii)), $P \cup \{\leftarrow$A$\}$ has an RSLD-refutation.

Now suppose that the result holds for $n-1$. Let A $\in$ TPE $\uparrow n$. By the definition of TPE, one of the following holds.

(i) There exists a ground instance of a clause B$\leftarrow$B$_1$, ..., B$_q$ such that A = Bθ and $\{$B$_1\theta$, ..., B$_q\theta\} \subseteq$ TPE $\uparrow n-1$, for some θ.

(ii) A is non-distinguished; there exists a ground instance of a clause solve(α)$\leftarrow$B$_1$, ..., B$_q$ such that $\uparrow$(Aθ) = $\alpha\theta$ and $\{$B$_1\theta$, ..., B$_q\theta\} \subseteq$ TPE $\uparrow n-1$, for some θ.

(iii) A is solve(M); there exists a ground instance of a clause B$\leftarrow$B$_1$, ..., B$_q$ such that $\uparrow$(Bθ) = Mθ and $\{$B$_1\theta$, ..., B$_q\theta\} \subseteq$ TPE $\uparrow n-1$, for some θ.

By the induction hypothesis, $P \cup \{\leftarrow$B$_i\}$ has a refutation, for i=1, ..., q. Since each B$_i$ is ground, these refutations can be combined into a refutation of $P \cup \{\leftarrow$(B$_1$, ..., B$_q$)$\theta\}$. Thus $P \cup \{\leftarrow$A$\}$ has an unrestricted RSLD-refutation and we can apply the extended lifting lemma to obtain an RSLD-refutation of $P \cup \{\leftarrow$A$\}$.■

The next theorem extends the result due to Hill, reported in [Lloyd 1987, theorem 8.4, p. 48], and is proved in the same way.

Theorem A.4. Suppose that $P \cup \{G\}$ is unsatisfiable. Then there exists an RSLD-refutation of $P \cup \{G\}$.

Proof. Let G be the goal $\leftarrow$A$_1$, ..., A$_k$. Since $P \cup \{G\}$ is unsatisfiable, G is false w.r.t. RMP. Thus $\{$A$_1\theta$, ..., A$_k\theta\} \subseteq$ RMP for some substitution θ. By theorem A.3, there is a refutation for $P \cup \{$A$_i\theta\}$, for i=1, ..., k. Since each A$_i\theta$ is ground, we can combine these refutations into a refutation for $P \cup \{G\theta\}$. Finally, we apply the extended lifting lemma.■

Lemma A.3. Let A be an atom. Suppose that $\forall$(A) is a logical consequence of P. Then there exists an RSLD-refutation of $P \cup \{\leftarrow$A$\}$ with the identity substitution as the computed answer.

Proof. Suppose that A has variables $x_1, \ldots, x_n$ (each of which can be indifferently either an object variable or a metavariable). Let $a_1, \ldots, a_n$ be distinct constants appearing neither in P nor in A such that the substitution $\theta = \{x_1/a_1, \ldots, x_n/a_n\}$ satisfies the substitution rules given in definition 4.1. Then it is clear that $A\theta$ is a logical consequence of P. Since $A\theta$ is ground, theorem A.3 shows that $P \cup \{\leftarrow A\theta\}$ has an RSLD-refutation. Since a_i's appear neither in P nor in A, by replacing a_i with x_i in this refutation, we obtain an RSLD-refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer.■

Now the main completeness result, originally due to Clark and reported in Lloyd (1987), theorem 8.6, p. 49), can be proved in a way similar to classical SLD-resolution.

Theorem A.5. (Completeness of RSLD-resolution) For every correct answer θ for $P \cup \{G\}$, there exist a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.

Proof. Suppose G is the goal $\leftarrow A_1, \ldots, A_k$. Since θ is correct, $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ is a logical consequence of P. By lemma A.3, there exists an RSLD-refutation of $P \cup \{\leftarrow A_i\theta\}$ such that the computed answer is the identity, for $i=1, \ldots, k$. We can combine these RSLD-refutations into an RSLD-refutation of $P \cup \{\leftarrow G\theta\}$ such that the computed answer is the identity.

Suppose that the sequence of mgus and/or gmgus of the RSLD-refutation of $P \cup \{\leftarrow G\theta\}$ is $\theta_1, \ldots, \theta_n$. Then $G\theta\theta_1 \ldots \theta_n = G\theta$. By the extended lifting lemma, there exists an RSLD-refutation of $P \cup \{\leftarrow G\}$ with mgus and/or gmgus $\theta_1', \ldots, \theta_n'$ such that $\theta\theta_1 \ldots \theta_n = \theta_1' \ldots, \theta_n'\gamma'$, for some substitution γ'. Let σ be $\theta_1' \ldots \theta_n'$ restricted to the variables in G. Then $\theta = \sigma\gamma$, where γ is an appropriate restriction of γ'.■