Reflection Principles in Computational Logic

JONAS BARKLUND, PIERANGELO DELL’ACQUA, Uppsala University, Computing Science Department, Box 311, S-751 05 Uppsala, Sweden. E-mail: jonas@csd.uu.se, pier@csd.uu.se

STEFANIA COSTANTINI, Department of Pure and Applied Mathematics, University of L’Aquila, via Vetoio Loc. Coppito, I-67100 L’Aquila, Italy. E-mail: stefcost@univaq.it

GAETANO A. LANZARONE, Center of Information Sciences, Faculty of Sciences at Varese, Insubria University, via Ravasi, 2 I-21100 Varese, Italy. E-mail: lanzarone@mail.varbio.unimi.it

Abstract

We introduce the concept of reflection principle as a knowledge representation paradigm in a computational logic setting. Reflection principles are expressed as certain kinds of logic schemata intended to capture the basic properties of the domain knowledge to be modelled. Reflection is then used to instantiate these schemata to answer specific queries about the domain. This differs from other approaches to reflection mainly in the following three ways. First, it uses logical instead of procedural reflection. Second, it aims at a cognitively adequate declarative representation of various forms of knowledge and reasoning, as opposed to reflection as a means for controlling computation or deduction. Third, it facilitates the building of a complex theory by allowing a simpler theory to be enhanced by a compact metatheory, contrary to the construction of metatheories that are only conservative extensions of the basic theory. A computational logic system for embedding reflection principles, called RCL (for Reflective Computational Logic), is presented in full detail. The system is an extension of Horn clause resolution-based logic, and is devised in a way that makes important features of reflection parametric as much as possible, so that they can be tailored according to specific needs of different application domains. Declarative and procedural semantics of the logic are described and correctness and completeness of reflection as logical inference are proved. Examples of reflection principles for three different application areas are shown. Relationship with a variety of distinct sources within the literature on relevant topics is discussed.

Keywords: Reflection, computational logic, knowledge representation paradigms, logical frameworks, logic programming.

1 Introduction

Reflective (or introspective, or self-referencing) systems have long been considered in many branches of logic and computer science, and more recently in their intersection area named computational logic or logic programming. Their importance and usefulness in logic [55, 56] and in theorem proving [38], in computer science [30, 51, 60], and in logic programming [7, 40, 47] has been generally recognized (see also [1, 11, 13, 32, 57] for snapshots of research).

The common intuitive notion of reflection in such different areas is that of an access re-
relationship between theories or programs at the object level and theories or programs at the metalevel. The object level is intended to represent knowledge about some domain, whereas the metalevel is intended to represent knowledge about the object level itself.

Though this basic notion manifests itself in a variety of degrees, forms and purposes in the work referenced above, in most cases the aim of the metalevel has been viewed as a guide for the object level inference or computation, i.e. ‘for expressing “properties of control” in the same way as “properties of the domain” ’ [62]. In this paper instead we take a different view, as we are concerned with expressing the abstract features and properties of a problem domain via (a general and powerful form of) reflection.

We present a logical system whose main objective is to allow its users to specify and experiment with a variety of deductive systems, given through axioms and rules of inference. The system is called RCL, standing for ‘Reflective Computational Logic’.

The syntax of the language (of the deductive systems that can be specified in RCL) is based on an enhanced Horn clause language, containing names for the expressions of the language itself. This makes it possible to specify deductive systems able to perform metareasoning and to represent knowledge and metaknowledge about a problem domain. The specification process is accomplished through the following four steps.

**Step I** In RCL, the first step for specifying a deductive system (DS) is that of defining its naming device (encoding). Encodings are formalized through equational theories (name theories). RCL leaves significant freedom in the representation of names. Therefore, users of RCL can explicitly make (to some extent) their own decisions about critical issues such as the representation of variables at the metalevel, or the choice of what syntactic entities to represent at the metalevel.

**Step II** After having defined (whenever necessary) a suitable naming convention, the user of RCL has to provide a corresponding unification algorithm that is able to handle names and to relate names to what is named.

**Step III** The third step is to represent the axioms defining the deductive system, DS, under consideration in the form of enhanced Horn clauses.

**Step IV** The last step for specifying DS is to represent the inference procedure.

In RCL, the specification of DS with its inference rules is executable, i.e. it can be directly used for deduction in DS. Moreover, the model-theoretic and fixed point semantics of DS are obtained as a side effect of the specification. Although reflection as a mean for extending logical theories has long been studied in the literature, the interpretation given here to this notion leads to a novel approach to defining and using new inference rules. In particular, the user is required to express an inference rule $R$ as a function $R$, called a reflection principle, from clauses, which constitute the antecedent of the rule, to sets of clauses, which constitute the consequent. Then, given a theory $T$ consisting of a set of initial axioms $A$ (enhanced Horn clauses) and of its deductive closure, and given a reflection principle $R$, a theory $T'$ containing $T$ is obtained as the deductive closure of $A \cup A'$, where $A'$ is the set of additional axioms generated by $R$. Consequently, the model-theoretic and fixed point semantics of $T$ under $R$ are obtained as the model-theoretic and fixed point semantics of $T'$. RCL, however, does not generate $T'$ in the first place. Rather, when queried about $DS$, RCL queries itself to generate the specific additional axioms usable to answer the query, according to the given reflection principles (i.e. according to the inference rules of DS). In Section 2, after a review of the relevant literature, we introduce the definition of reflection principle.
In order to exhibit this intended behaviour, RCL is built as a self-referential, reflective system, procedurally based on an extended resolution principle that implements reflection. The RCL system that we present falls within the logic programming approach. In fact, it extends the language of Horn clauses with the kind of facilities mentioned above, and extends the well-established semantics and proof theory of Horn clauses accordingly. We believe, however, that the underlying ideas could find application also in in the context of other formalisms.

We intend to show that the proposed system is a practical, principled and powerful computational logic system.

The system is practical in that it gives its users two flexible tools to construct their own representation and deduction forms rather than providing specific ones.

For representation, as mentioned above, specific encoding and substitution facilities are not predefined and built into the system; rather, the system allows them to be user-defined by means of name theories, i.e. sets of equational axioms with associated rewrite systems. The expressive power of encodings can therefore be traded against (computational and semantic) properties enjoyed by the associated rewrite systems in a maximally flexible fashion, in order to tailor the system to the application domain at hand. This is introduced and discussed in Section 3.

Then, the integration of reflection principles into the declarative and procedural semantics of Horn clause theories is discussed in Sections 4.1 and 4.2.

The system is principled because its semantics and proof theory are formally defined in a way that is not a departure from classical Horn clause logic, as shown in Sections 4.1 and 4.2. Results of soundness and completeness of the proof theory with respect to the model theory are given in Section 5.

The system is powerful in a twofold sense. First, it is usually easier to represent domain knowledge by first considering an initial core theory and then reflectively extending it by means of reflection principles, than to consider the whole theory all at once from the beginning. Second, and perhaps more important, reflection principles are epistemologically suitable for representing basic abstract properties of a problem domain, especially for some complex domains and sophisticated application areas. We argue in favour of this view in Section 6, where three domains are exemplified and treated as case studies.

The first deductive system that we define (Section 6.1) is a metalogic programming language, Reflective Prolog, that provides: (i) names, (ii) the possibility of defining knowledge on multiple levels, and (iii) the possibility of exchanging knowledge between levels by means of a distinguished reflective predicate. Precisely, there is a certain predicate \( p \) in the language such that, for a class of formulae \( f \) of the language itself, the formula \( px(\overline{f^n}) \rightarrow f \) is true (where \( \overline{f^n} \) denotes the encoding of \( f \), and \( px \) denotes predicate \( p \) in the context of a substitution facility to replace variables \( x \) of \( f \)). \( p \) is called a reflective predicate, and is to be defined as an approximation of a truth or proof predicate. The approximation has to be such that the intended useful features of self-reference are obtained, without running into the well-known paradoxes (see Perlis [55] for a discussion). We will show that two reflection principles are able to model the behaviour of a reflective predicate.

The second deductive system is able to represent agents and cooperation between multiple agents (Section 6.2). In particular, we consider rational agents that are introspective and communicative. A simple reflection principle models a quite general form of inter-agent communication.

The third deductive system (Section 6.3) is aimed at performing analogical reasoning. It
is able to model a source domain representing knowledge which is certain and complete, and a target domain where knowledge is either uncertain or incomplete. Assuming that it can find in the target domain some knowledge which is analogous to corresponding knowledge in the source domain, this deductive system is able (via a simple reflection principle) to apply analogy in performing deduction, thus drawing conclusions in the target domain which would have been impossible and incorrect to derive without the analogy.

A main aim of this research is that of defining a well-founded theoretical framework, but also a foundation for a practically implemented system: practical feasibility has been taken into account while defining all the aspects of the approach. Currently, an actual implementation is being designed, written in Reflective Prolog (of which a complete implementation exists, based on a Prolog meta-circular interpreter). The implementation is planned to have the following features: a default encoding device and some default reflection principles are included. The system however is intended to be parametrical w.r.t. these two components. Thus, the implementation will be adaptable to a specific application domain by replacing the encoding device, and/or adding new reflection principles. In this case however, the implementation of these components is to be provided (in most cases by modifying the default one), along the lines specified in the paper (rewriting system for the encoding, extended resolution reflection principles). The system and its implementation are devised so that the components to be modified/extended are suitably encapsulated, in order to be easily managed, with limited risk of introducing unintended malfunctioning. We can say that, from a theoretical point of view, RCL is a framework for defining new deductive systems, and from a practical point of view, its basic implementation should be a ‘toolkit’ for easily obtaining the construction of these new systems.

The paper is concluded in Section 7, where we discuss the scope of the proposed approach and its limitations, examine areas of possible applications, and review previous work in the literature and possible relationships to ours. Proofs of theorems are given in the Appendix.

2 Reflection and reflection principles

2.1 Background

In this section we recall the concepts we will be introducing and discussing, giving a basic historical background and perspective of the state of the art on these topics (the reader may also refer to [1, 3, 13, 32, 50, 57] for an overview.)

A computational system is a system that reasons and acts upon some domain. The system represents (some of the features of) its domain under the form of data, and prescribes how these data should be manipulated. The system is causally connected (to its domain) if the system and the domain are linked in such a way that any change in one of the two leads to some effect upon the other. A system controlling a robot arm is a typical example of a causally connected system. This system may incorporate data representing the position of the arm. This data changes whenever the arm is moved by some external force; vice versa, if the system changes this data, then the robot arm changes to the corresponding position.

A metasystem is a system that has as domain another computational system, called object system, and has a representation (at the metalevel) of the features of the object system as data. Many examples of metasystems can be mentioned; an example is a compiler that is able to compile itself. Another well-known example is the notion of meta-interpreter and partial evaluation, as used in Lisp and Prolog. A Prolog program, for instance, may incorporate a de-
fault meta-interpreter that simulates at the metalevel (some of) the features of the underlying interpreter. Note that meta-interpreters are written in the same language as the program they interpret. The default meta-interpreter can be used as a basis for building a special-purpose meta-interpreter. Meta-interpreters can be used, for instance, in implementing: (i) variants of the language; (ii) enhanced control strategies; (iii) analysis and debugging tools; and (iv) auxiliary inference strategies, related to the application domain of the program at hand. The meta-interpreter can then be partially evaluated and compiled together with the program it is designed for, thereby producing a special-purpose interpreter.

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 architectures (see [3, 70] for a comprehensive overview and classification). Recognized advantages of metalevel representations are in the possibility of separation between domain knowledge and control knowledge and of better mastering inference by explicit treatment of control. 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.

If a metasystem which acts on a representation of its own features is causally connected (to itself), then it is able to reflect or introspect, i.e. it is able to manipulate data representing itself in compliance with its semantics. Causal connection in this case means that the representation is linked in a consistent way to the represented objects.

The first reflective system to appear in the literature is (to the best of our knowledge) the FOL system by Weyrauch [73]. In FOL, knowledge and metaknowledge are expressed in different contexts, and the user can access them both for expressing and inferring new facts. A FOL context consists of a language \( L \) (which is a first-order language with sorts and conditional expressions) and a simulation structure \( S \), which is a partial finite representation of some model. Causal connection is guaranteed by means of attachments, which are user-defined explicit definitions relating symbols in \( L \) with their interpretation in \( S \). A special context named META describes the proof theory and some of the model theory of a FOL context \( \mathcal{V} \) whose metatheory is META. The connection between \( \mathcal{V} \) and META is established by the attachments and by a special linking rule that is applicable in both directions:

\[
\begin{align*}
\text{Theorem}(\overline{W}) \\
\hline
W
\end{align*}
\]

where \( W \) is any formula in the theory, \( \overline{W} \) is a representation (a name) for \( W \) and \( \text{Theorem}(\overline{W}) \) is a fact in the metatheory. By means of a special primitive, called Reflect, the linking rule can be explicitly applied by the user. Its effect is either of reflecting up a formula \( W \) to the metatheory, so as to derive metatheorems involving \( \overline{W} \), or of reflecting down a metatheorem \( \overline{W} \), so that \( W \) becomes a theorem of the theory. Metatheorems can therefore be used as subsidiary deduction rules. The consistency and correctness of the application of Reflect is left to the user, as is the whole mechanism of contexts with attachments.

A seminal approach to reflection in the context of Horn clause language is MetaProlog, proposed by Bowen and Kowalski [12]. They propose to describe Horn clause syntax and provability in the logic itself by means of a careful version of the default meta-interpreter (specified via a predicate \( \text{Demo} \) that is defined by a set of axioms \( \text{Pr} \)), where all these aspects are made explicit. Also in this case, the connection between the object level and the metalevel
is provided by linking rules for up and down reflection:

\[
\frac{T \vdash_L A}{\Pr \vdash_M \text{Demo}(\Gamma T \models, \Gamma A \models)} \quad \frac{\Pr \vdash_M \text{Demo}(\Gamma T \models, \Gamma A \models)}{T \vdash_L A}
\]

where \( \vdash_M \) and \( \vdash_L \) mean provability at the metalevel and at the object level, respectively, \( T \) is a Horn clause theory and \( A \) is an object level formula. As this approach is based on meta-interpretation, the object language and the metalinguage are of course the same or, according to a popular terminology, they are amalgamated. In fact, the approach allows mixed object level and metalevel rules. Again, the application of the linking rules (which coincides, in practice, with the invocation of Demo) is left to the user, i.e. reflection is explicit. The semantics of this approach is, however, not easy to define [41, 26, 48, 53, 64], and holds only if the metatheory and the linking rules provide an extension to the basic Horn clause language which is conservative, i.e. only if Demo is a faithful representation of Horn clause provability. Although the amalgamated language is far more expressive than the object language alone, enhanced meta-interpreters are (semantically) ruled out, since in that case the extension is non-conservative. This excludes the possibility of using Demo for expressing auxiliary deduction rules in a semantically sound way.

The amalgamated approach has also been experimented by Attardi and Simi in Omega [5]. Omega is an object-oriented formalism for knowledge representation, which can deal with metatheoretical notions by including objects that describe Omega objects themselves and derivability in Omega.

3-Lisp [61] is another important example of an amalgamated reflective architecture. 3-Lisp is a meta-interpreter for Lisp, or, more precisely, a metacircular interpreter that represents explicitly not only the control aspects, but also the data structures of the underlying interpreter. Here, the metalevel is accessible from the object level at run-time through a reflection act. The program is able to interrupt its computation, to change something with its interpretation, and to continue with a modified interpretation process. This kind of mechanism is called computational reflection. The semantics of computational reflection is procedural, however, rather than declarative. A reflective architecture conceptually similar to 3-Lisp has been proposed for the Horn clause language and has been fully implemented [15].

A non-amalgamated approach in logic programming is Gödel [42] (object theory and metatheory are distinct). Gödel also provides a (conservative) provability predicate, a partial evaluation facility and an explicit form of reflection.

A project that extends and builds on both FOL and 3-Lisp is Getfol [34, 36]. It is developed on top of a reimplementaiton of FOL (therefore the approach is not amalgamated: the object theory and metatheory are distinct). Getfol is able to introspect its own code (lifting), to reason deductively about it in a declarative metatheory and, as a result, to produce new executable code that can be pushed back to the underlying interpretation (flattening). The architecture is based on a sharp distinction between deduction (FOL style) and computation (3-Lisp style). The main objective of Getfol seems to be that of implementing theorem-provers, given its ability of implementing flexible control strategies to be adapted (via reflection) to the particular situation. Similarly to FOL, the kind of reasoning performed in GETFOL consists in: (i) performing some reasoning at the metalevel; (ii) using the results of this reasoning to assert facts in the object level. An interesting extension is, however, that of applying this concept to a system with multiple theories and multiple languages (each theory formulated in its own language) [35], where the two steps are reinterpreted as (i) doing some reasoning in one theory and (ii) jumping into another theory to do some more reasoning on the basis
of what has been derived in the previous theory. These two deductions are concatenated by
the application of *bridge rules*, which are inference rules where the premisses belong to the
language of the former theory, and the conclusion belongs to the language of the latter.

From the point of view of semantics, we may notice that an explicit reflection that ex-
tends the inference relation of the object level disturbs the (classical) object level semantics:
by downward reflection, facts and/or formulas are added that are not logically entailed by
the available object level knowledge. In order to face this problem, Hoek *et al.* [68] and
Treuer [67] adopt temporal logics and epistemic states of knowledge. Moreover, metalevel
computation may in general be costly [69], and explicit reflection certainly is a potential
source of inefficiency (especially whenever it is based on some form of meta-interpretation).
With explicit reflection and inefficient metalevel computation, metalevel knowledge will most
often play a secondary role w.r.t. object level knowledge.

To overcome these problems, a different concept of reflection has been incorporated into
Reflective Prolog [19, 22], a self-referential Horn clause language with logical reflection. The
objective of this approach was that of developing a more expressive and powerful language,
while preserving the essential features of logic programming: Horn clause syntax, model-
theoretic semantics, resolution via unification as procedural semantics, correctness and com-
pleteness properties. To investigate the relation between this kind of logical reflection and
the corresponding model-theoretic semantics, an interpreter of Reflective Prolog has been
fully implemented [27]. In Reflective Prolog, Horn clauses are extended with self-reference
and resolution is extended with logical reflection, in order to achieve greater expressive and
inference power. The reflection mechanism is *implicit*, i.e. the interpreter of the language
automatically reflects upwards and downwards. This allows reasoning and metareasoning to
interleave without the user’s intervention, so as to exploit both knowledge and metaknowl-
edge in proofs (in most of the other approaches, instead, there is one level which is ‘first–
class’, where deduction is actually performed and the other level which plays a secondary
role). The reflection mechanism is embedded in both the procedural and the declarative se-
mantics 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. Proce-
durally, this implies that there is no need to axiomatize provability in the metatheory. Object
level reasoning is not simulated by meta-interpreters but directly executed by the language
interpreter, thus avoiding unnecessary inefficiency. The formal semantics is defined in cor-
respondence to the behaviour of the interpreter: a theory composed of an object level and
(one or more) metalevels is semantically regarded as an enhanced theory, enriched by new
axioms which are entailed by the given theory and by the linking rules interpreted as axiom
schemata. Therefore, in Reflective Prolog, language and metalanguage are amalgamated in a
non-conservative extension, though avoiding semantic problems. Reflective Prolog has been
proposed as an enhanced knowledge-representation language [24].

In order to compare Getfol and Reflective Prolog, as recent fully implemented systems, we
may note that:

- reflection in Getfol gives access to a metatheory where many features of the system are
  made explicit, even the code that implements the system itself. In contrast, reflection in
  Reflective Prolog gives access to a metatheory where various kinds of metaknowledge can
  be expressed, either about the application domain or about the behaviour of the system;
- deduction in GETFOL consists in performing some reasoning at the metalevel and then
  asserting facts at the object level. Deduction in Reflective Prolog means using at each step
  either metalevel or object level knowledge, in a continuous interleaving between levels:
Reflection Principles in Computational Logic

i.e. both levels are ‘first–class’ in the deductive process;
• metareasoning in Getfol implies defining explicit syntactic manipulation of descriptions, while metareasoning in Reflective Prolog implies a declarative definition of metaknowledge, which is automatically integrated into deductions. This corresponds to the different aims of the two systems: theorem-proving for Getfol and knowledge representation for Reflective Prolog.

2.2 The concept of reflection principle

The idea of reflection in logic dates back to work by Feferman [31]. He introduced the concept of a reflection principle defined as:

‘a description of a procedure for adding to any set of axioms \( A \) certain new axioms whose validity follow from the validity of the axioms \( A \) and which formally express within the language of \( A \) evident consequences of the assumption that all the theorems of \( A \) are valid.’

Thus, in Feferman’s view, reflection principles do not generate arbitrary consequences, but rather a transposition of the original ones. In RCL, we reinterpret the concept of a reflection principle as:

‘a description of a procedure for adding to any set of axioms \( A \) certain new axioms whose validity follow from some user-defined inference rules.’

We use reflection principles to integrate into the (declarative and procedural) semantics of the Horn clause theories the inference rules of a deductive system \( DS \) that a user wants to define. Inference rules are, by definition, decidable relations between formulae of a language \( L \), and can be expressed in the form of axiom schemata. These schemata need however to be given a role in the theory, both semantically (obtaining a semantics for the resulting theory) and syntactically (making them usable in deduction). We choose to interpret them as procedures, more precisely as functions that transform Horn clauses into (sets of) Horn clauses. These new Horn clauses are called ‘reflection axioms’. Thus, the difference with respect to Feferman’s notion of reflection principles is that the validity of the reflection axioms is not necessarily a formal consequence of the validity of the given axioms. In fact, a user could define also non-standard inference rules, to encode various forms of uncertain or plausible reasoning. Nevertheless, in a given application context, where a new inference rule is introduced to capture some specific aspects of the domain under consideration, the validity of reflection axioms should follow conceptually, according to the intended meaning of the extension.

The advantage of representing inference rules in the form of reflection principles is that the model-theoretic and fixed point semantics of the given theory under the new inference rule coincides with the corresponding semantics [44] of the plain Horn clause theory obtained from the given theory, augmented by the reflection axioms.

The advantage of applying reflection principles on a single clause is that the reflection axioms need not be generated in the beginning, but can be generated dynamically, whenever a reflection principle is applicable to the input clause of any resolution step.

In this and the following sections, we present a formalization of the proposed concept of reflection which should constitute a simple way of understanding reflective programs as well as a description of how reflection allows one to uniformly treat different application
Reflection Principles in Computational Logic 751

areas. The applications of reflection that we have previously studied (and reported in detail elsewhere [20, 24, 25]) are instances of the new formalization. Thus we are able to present them as case studies and show how RCL can constitute a uniform framework for several problem domains.

For each of those areas, we present the reflection principles suitable to capture the specificity of the problem domain. Given a basic theory expressing a particular problem in that domain, its extension determined by the chosen reflection principle contains the consequences intended by that principle, but not entailed by the basic theory alone. Thus, this use of reflection is different in essence from previous use of reflection rules in logic programming, such as in [12]. Our conception and use of reflection principles are precisely aimed at making the set of theorems that are provable from the basic theory, augmented with reflection axioms, differ from the set of theorems that are provable from the basic theory alone. This capability allows one to model several forms of reasoning within the same formal framework. The version of RCL presented in this paper is monotonic, in the sense that reflection principles enlarge the set of consequences of the basic theory. The use of reflection in non-monotonic reasoning is discussed by Costantini and Lanzarone [23]. Their approach can be integrated in RCL (at the expense of some semantic complications).

Notice that reflection principles allow one to formalize how conclusions follow one from the other, not necessarily between an object theory and a metatheory (in the latter case you need an encoding device). Reflection principles express inference rules to be applied within the same theory, or even to link different theories (similarly to the bridge rules of Giunchiglia and Serafini [35]).

**Definition 2.1**

Let $C$ be a definite clause. A reflection principle $\mathcal{R}$ is a mapping from definite clauses to (finite) sets of definite clauses. The clauses in $\mathcal{R}(C)$ are called reflection axioms.

Given a definite program $P = \{ C_1, \ldots, C_n \}$, we write $\mathcal{R}(P)$ for $\mathcal{R}(C_1) \cup \ldots \cup \mathcal{R}(C_n)$. The following example, although very simple, informally illustrates the main idea.

**Example 2.2**

Suppose we want to incorporate into a theory $T$ the ability to reason about ‘provability’ in the theory itself. To do this, we can introduce a predicate $\text{demo}$ defined over representations of propositions in $T$ itself, such that $\text{demo}$ holds for all those representations for which the corresponding propositions are provable. This can be formalized as:

$$\frac{\alpha_i}{\text{demo}(\alpha_i)}$$

where $\alpha_i$ indicates the name of $\alpha_i$. Thus, $\text{demo}(\alpha_i)$ is provable in the theory whenever proposition $\alpha_i$ is. In RCL, the inference rule above can be incorporated by means of the following reflection principle $\mathcal{R}$:

$$\mathcal{R}(\alpha_i) = \{ \text{demo}(\alpha_i) \leftarrow \alpha_i \}.$$  

Assume that $T$ contains the following initial set of axioms:

$$A = \{ \alpha_1, \alpha_2, \alpha_3 \leftarrow \text{demo}(\alpha_2) \}.$$  

Then, the set $A'$ of reflection axioms generated by $\mathcal{R}$ is:

$$A' = \mathcal{R}(A) = \{ \text{demo}(\alpha_1) \leftarrow \alpha_1, \text{demo}(\alpha_2) \leftarrow \alpha_2, \text{demo}(\alpha_3) \leftarrow \alpha_3 \}.$$  

752 Reflection Principles in Computational Logic

The deductive closure of $A \cup A'$ is the theory:

$$T' = \{\alpha_1, \alpha_2, \alpha_3, demo(\alpha_1), demo(\alpha_2), demo(\alpha_3)\}.$$

Notice that several reflection principles can co-exist in the same framework. This is the case of the application outlined in Section 6.1.

Reflection principles allow extensions to be made to the language of Horn clauses by modifying the program but leaving the underlying logic unchanged. A potential drawback is that the resulting program $(P \cup R(P), E)$ may have, in general, a large number of clauses, which is allowed in principle but difficult to manage in practice. To avoid this problem, reflection principles are applied in the inference process only as necessary, thus computing the reflection axioms ‘on the fly’. (This means that we do not create $A'$ or $A \cup A'$ explicitly.)

Given a reflection principle $\mathcal{R}$, we hereafter write $\Delta_{\mathcal{R}}$ to indicate any procedure that is able to compute $\mathcal{R}$. It is important to notice that $\Delta_{\mathcal{R}}$ can be any suitable formal system for the application at hand. In particular, $\Delta_{\mathcal{R}}$ may be a metaprogram in some metalogic language. In RCL, whenever its users define a reflection principle $\mathcal{R}$, they must provide $\Delta_{\mathcal{R}}$, and they are responsible for it being a correct implementation of $\mathcal{R}$.

The antecedent of the inference rule expressed as a reflection principle being a single Horn clause is not really a limitation. In fact, by defining a suitable name theory, the given clause may encode any set of formulae. The consequent being a set of Horn clauses is an actual limitation. In fact, in this sense RCL is not a departure from the traditional logic programming approach, as user-defined inference rules can express only what can be expressed (either at the object level or at the metalevel) by means of Horn clauses.

3 The enhanced Horn clause language

The distinction between use and mention of a term, or between language and metalanguage, and the technique of giving names to language expressions in order to be able to talk about their properties, both belong to the tradition of philosophical and mathematical logic.

Since our aim is to devise a language that is both cognitively adequate and practically usable, in this section we first motivate the use of names and then introduce the technicalities by which they can be defined in a suitable and flexible way.

Notice that giving names to language expressions is the only way to have both language and metalanguage while staying within first-order logic, which is a strongly desirable property in a computational setting.

In the following, a ‘ground term’ is a term not containing variables. Consequently, different approaches to giving names to expressions can be divided into ‘ground’ naming approaches, where names are ground terms, and ‘non-ground’ naming approaches, where names are terms that may contain variables.

3.1 Use and mention

In a language there is a clear distinction between a thing and its name: we use names to talk about things. However, when we want to mention expressions, rather than using them, confusion can arise (see Suppes [65] for a discussion on this topic). Consider the following statements:

*California is a state.*
California has ten letters. \hspace{1cm} (3.2)

‘California’ is a state. \hspace{1cm} (3.3)

‘California’ has ten letters. \hspace{1cm} (3.4)

The statements (3.1) and (3.4) are true, while (3.2) and (3.3) are false. To say that the state-name in question has ten letters we must use not the name itself, but a name of it. The name of an expression is commonly formed by putting the named expression between quotation marks. The whole, called a *quotation*, denotes its internal content. This device is used, for example, in statement (3.3). Every name denotes a thing. For example, California denotes the well-known American state. Names of things can also be seen as things themselves denoted by other names (i.e. quotations), like ‘California’. The reading of statement (3.3) can be clarified by rephrasing it as:

*The word ‘California’ is a state.*

(3.3) is about a word which (3.1) contains, and (3.1) is about no word at all, but a state. In (3.1) the state-name is *used*, while in (3.4) a quotation is used and the state-name is *mentioned*. To mention California we use ‘California’ or a synonym, and to mention ‘California’ we use ‘‘California’’ or a synonym.

We could also baptise the word ‘California’ with a personal name. Let

\[ \textit{Jeremiah} = ‘\textit{California}’. \hspace{1cm} (3.5) \]

Then the following statements could be true,

\[ \textit{Jeremiah} \textit{is a name of a state}. \hspace{1cm} (3.6) \]
\[ \textit{Jeremiah} \textit{has ten letters}. \hspace{1cm} (3.7) \]
\[ ‘\textit{Jeremiah}’ \textit{has eight letters}. \hspace{1cm} (3.8) \]

while the next is false

\[ ‘\textit{Jeremiah}’ \textit{is a name of a state}. \hspace{1cm} (3.9) \]

Statement (3.9) could be rendered true by inserting another ‘name of’ in it

\[ ‘\textit{Jeremiah}’ \textit{is a name of a name of a state}. \]

Thus, by quoting an expression we can ascribe different kinds of properties to it: for example, morphological properties as in statement (3.4) or phonetic and grammatical properties as in the following.

\[ ‘\textit{Boston}’ \textit{is disyllabic}. \hspace{1cm} (3.10) \]
\[ ‘\textit{Boston}’ \textit{is a noun}. \hspace{1cm} (3.11) \]

We can also ascribe *semantic properties*, that is, properties that arise from the meaning of the expression.

\[ ‘\textit{Boston}’ \textit{designates the capital of Massachusetts}. \hspace{1cm} (3.12) \]
\[ ‘\textit{Boston}’ \textit{is synonymous with ‘the capital of Massachusetts’}. \hspace{1cm} (3.13) \]
Notice that in (3.13) quotations can be synonymous, while places cannot.

As Quine points out [59], the use of quotation marks is the main practical measure against confusing objects with their names. Frege was the first logician to use quotation marks formally to distinguish use and mention of expressions (see Carnap [14] for further discussion).

Quotations can also be applied to non-atomic expressions. For example, to say that a statement has a given property, e.g. the semantic property of truth or falsehood, we attach the appropriate predicate to the name of the statement in question, and not to the statement itself. Thus, we may write:

\[ \text{'Margus is Estonian' is true.} \] (3.14)

but never

\[ \text{Margus is Estonian is true.} \] (3.15)

(3.14) is a statement, while (3.15) is not. Notice that in (3.14) we use a predicate to speak about another statement, therefore we mention it. In contrast, logical connectives attach to statements (and not to names of statements) to form more complex statements, and this application can be iterated.

Quantifiers standing outside of quotes cannot bind variables occurring inside quotes because by quoting a variable we mention it. Consider the following statement:

\[ \text{For every } p, \ 'p' \text{ is the sixteenth letter of the alphabet.} \] (3.16)

This sentence can be considered to be true, and the quantifier \( \text{For every } p \) to be redundant and not binding the occurrence of \( p \) inside the quotes. In contrast, if we were to regard the quantifier as binding the occurrence of \( p \) in quotes, we would obtain, replacing \( p \) by \( \text{Margus is Estonian} \), the false assertion:

\[ \text{'Margus is Estonian' is the sixteenth letter of the alphabet.} \] (3.17)

Tarski [66], for example, defines names as variable-free terms. He discusses two kinds of names: quotation-mark (or primitive) and structural descriptive names. The former category associates with a formula a ‘monolithic’ term as its name (Gödel’s encoding is an example of this kind of naming). The latter category associates with a formula a structured ground term that reflects the structure of the sentence it names. The advantage of structural descriptive names over quotation-mark names is that they allow us to quantify over parts of expressions.

Names have been widely used in computational logic. In a formal language, we can have names of formulae, but also, more generally, names of elements of the language that we can call expressions. The association between expressions and names is usually called a naming relation. The domain of a naming relation is a subset of the set of all language expressions, and possibly includes predicate, function and variable symbols, terms, atoms, single formulae as well as sets of formulae. Theories in the language may also have names. Some expressions may have primitive names, some others structural descriptive ones. In principle, an expression may have more than one name. In practice, however, naming relations are typically functional and injective (see van Harmelen [71] for a discussion on the properties of naming relations).

A name is itself an expression in a formal language. The operation which results in obtaining the name of an expression (or, more generally, in relating a name with what it names) has
been called *quotation*, or *referentiation*, or *reification*, or *encoding*. The converse operation is usually called *unquotation* or *de-referentiation*. When expressions that define names are terms of a language, they are called *name terms*. Whenever names of expressions in a given formal language are expressed in the language itself, i.e. whenever a language is capable of self-reference, we call it a *metalogic language*. A theory expressed in a metalogic language therefore consists of the *object level*, composed of object formulae not containing name terms and of the *metalevel*, consisting of formulae containing name terms. Formulae of the metalevel express some kind of syntactic or semantic properties of object formulae (as outlined in the simple examples above), and thus express some kind of *metaknowledge*, that can be used in deduction in various ways, thus performing *metareasoning*. The reader may refer to [2, 3, 24] for a discussion about possible uses of metaknowledge and metareasoning.

Notice that it is somewhat controversial whether a language should be capable of self-reference, or if names should be encoded in a separate metalanguage. As discussed in Section 2.1, the two points of view have led to systems where the object and the metalevels are separated (e.g. FOL, GETFOL, Gödel) or amalgamated (e.g. 3-Lisp, MetaProlog, Reflective Prolog). In our opinion, and in our experience, amalgamated approaches are in order for applications in knowledge representation, where expressing knowledge about a domain means also expressing properties which can be seen in one perspective as properties of the domain, and in another perspective as (syntactic metalevel) properties of knowledge itself.

In the next section, we will first extend the Horn clause language to a more general language able to express name terms. Then, we will show how user-defined naming relations can be expressed by means of the axioms of an equality theory. With this aim, we will consider some examples taken from the recent literature; in fact, we will show how to define the encoding used in some existing metalogic languages. Finally, we will consider how to extend unification so as to accommodate names. In this direction, we consider the formalization of the unification algorithm in terms of a rewrite system and then show how to extend this rewrite system to cope with equality theories defining names.

### 3.2 A metalanguage

We extend the language $HC$ of Horn clauses to an enhanced language $HC^+$ containing names of the expressions of the language itself. As we will see, $HC^+$ allows significant freedom in the choice of names: we only require that names of compound expressions be *compositional*, i.e. that the name of a compound expression must be obtained from the names of its components. In this language, it is possible to express various forms of encoding, both *ground* and *non-ground*, each of them with an associated rewrite system. We remind the reader that in a *ground* representation each syntactic expression is represented by means of a ground term. In contrast, *non-ground* representations do not require groundness of names.

The language is that of definite programs, as defined by Lloyd [49], except that terms are defined differently, in order to include *names* (called *name terms*) that are intended to represent the symbols and the expressions of the language itself.

The alphabet of $HC^+$ differs from the usual alphabet of definite programs by making a distinction between variables and *metavariables* and through the presence of *metaconstants*. Only names of $HC^+$ can be substituted for metavariables. Metaconstants are intended as names for constants, function symbols, predicate symbols and metaconstants themselves. If $c$ is a constant, a function or a predicate symbol in $HC^+$, then we write $c^1$ as a convenient notation for the metaconstant that names $c$ in $HC^+$. Similarly, if $c^n$, with $n > 0$, is a
metaconstant (i.e. \( c^n \) is \( c \) named \( n \) times), then its name is written as \( c^{n+1} \). Furthermore, the alphabet of \( HC^+ \) contains two operators, \( \uparrow \) and \( \downarrow \), and a distinguished predicate symbol, \( = \). The operators \( \uparrow \) and \( \downarrow \) are intended to denote the operations of quoting and unquoting, respectively. The symbols \( \uparrow, \downarrow, \) and \( = \) play a special role in the extended SLD-resolution and we assume that there are no symbols naming them.

Where not otherwise stated, the lower-case characters \( x, y \) and \( z \) (possibly indexed) are used for variables, while the upper-case characters \( X, Y \) and \( Z \) (possibly indexed) are used for metavariables. Thus \( x \) and \( y \), for example, are variables, and \( Z \) and \( X_3 \) are metavariables. Sometimes, to abbreviate the notation of expressions we use the notation reserved for variables to indicate both variables and metavariables, and we explicitly state this use.

The definition of terms \( (T) \) in \( HC^+ \) extends the usual one to contain name terms \( (NT) \) as a subset. Name terms contain metaconstants and metavariables, as well as names of compound expressions. We write the name of a compound expression of the form \( \alpha_0(\alpha_1, \ldots, \alpha_n) \) in \( HC^+ \) as \( \langle \alpha_0, \beta_1, \ldots, \beta_n \rangle \), where each \( \beta_i \) is the name of \( \alpha_i \), with \( 0 \leq i \leq n \). Furthermore, the name of the name of \( \alpha_0(\alpha_1, \ldots, \alpha_n) \) is the name term \( \langle \gamma_0, \gamma_1, \ldots, \gamma_n \rangle \), where each \( \gamma_i \) is the name of \( \beta_i \), with \( 0 \leq i \leq n \), etc. Requiring names of compound expressions to be compositional allows us to use unification for constructing name terms and accessing their components. Given a term \( t \) and a name term \( s \), we write \( \uparrow t \) to indicate the result of quoting \( t \) and \( \downarrow s \) to indicate the result of unquoting \( s \).

If we want to express properties (metaknowledge) of an expression of the object language (that expresses knowledge) such as \( p(a, b) \), we have to employ a name of that expression, represented here as \( \langle p^1, a^1, b^1 \rangle \), where \( p^1 \) is the metaconstant that names the predicate symbol \( p \), while the metaconstants \( a^1 \) and \( b^1 \) name the constants \( a \) and \( b \), respectively. We may, for example, express that \( p \) is a binary predicate symbol as \( \text{binary_pred}(p^1) \). Notice that we have employed the name of \( p \) and not \( p \) itself because we express something about the predicate symbol \( p \) (and a predicate symbol cannot appear in a term position).

We now present the definitions of definite programs, equality theories and logic programs. Let \( p \) be an \( n \)-ary predicate symbol distinct from \( = \), and let \( t_1, \ldots, t_n \) be terms. Then \( p(t_1, \ldots, t_n) \) is an atom and \( t_1 = t_2 \) is an equation. A name equation is an equation that contains at least one occurrence of \( \uparrow \) or \( \downarrow \). Observe that atoms and equations are distinct. An equality theory is a (possibly infinite) set of equations. Let \( A \) and \( A_1, \ldots, A_m (m \geq 0) \) be atoms not containing any occurrence of \( \uparrow \) and \( \downarrow \), and let \( e_1, \ldots, e_q (q \geq 0) \) be equations. Then \( A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m \) is a definite clause. If \( m = 0 \), then the clause is called a unit clause. A definite program is a finite set of definite clauses. A definite goal is a clause of the form \( \leftarrow A_1, \ldots, A_k \), with \( k \geq 0 \).\(^1\) If \( P \) is a definite program and \( E \) an equality theory, then \( (P, E) \) is a logic program. \( E \) contains axioms characterizing \( = \) (for example the usual equality interpretation of \( = \{[16]\} \), and \( P \) defines the meaning of the non-logical symbols.

\(^1\)All our clauses and goals will be definite and so we will omit ‘definite’ from now on.
The language $HC^+$

In the figure above, $\alpha^*$ denotes a (possibly empty) sequence of $\alpha$s and $\alpha^+$ denotes a non-empty sequence of $\alpha$s.

What we need now is a way to formalize the relation between terms and the corresponding name terms. We do this by formalizing the intended role of the operators $\uparrow$ and $\downarrow$ through equational theories that are a parameter of $RCL$.

**Example 3.1**

Often it is useful to access information as a sequence of characters, represented in the program as a constant. In Prolog, for example, there is a built-in predicate, `name`, that relates constants and their ASCII encodings.

There are two typical uses of `name`: (i) given a constant, break it down into single characters, (ii) given a list of characters, combine them into a constant. An example of a first kind of application would be a predicate that is true when a constant starts with a certain character. This may be defined in $HC^+$ as:

\[
P = \{ \text{starts}(x, y) \leftarrow X = \uparrow x, Y = \uparrow y, \text{first}_\alpha(X, Y) \}
\]

\[
E = \{ \uparrow a = 97, \uparrow b = 98, \uparrow z = 122, \uparrow c_1 \cdots c_n = [\uparrow c_1, \ldots, \uparrow c_n] \text{ for every constant } c_1 \cdots c_n \}
\]

where `starts(x, y)` holds if the constant $x$ starts with the character $y$ and the atom `first_\alpha(X, Y)` holds if $Y$ is the first element of the list $X$. The equality theory $E$ formalizes the relation between constants and their ASCII encodings. The axiom $\uparrow c_1 \cdots c_n = [\uparrow c_1, \ldots, \uparrow c_n]$ in $E$ is an axiom schema for any constant of the form $c_1 \cdots c_n$. 
3.3 Formalizing encodings

In order to name in $HC^+$ expressions of the language itself we employ an encoding. Encodings can represent various kinds of information: syntactic information, computational information, epistemological information, etc. (for an overview of encodings, cf. van Harmelen [72]). In general it is not possible to find an encoding that is optimal for all metalevel theories. This is because the syntactic richness of the encoding determines not only the expressivity of the metatheory, but also its complexity. Therefore, the encoding should be adapted to the particular requirements of a given metatheory, and/or to the application domain at hand. This motivates the choice, made in our formal framework, to provide the encoding as a separately definable component.

With this aim, encodings can be expressed by means of equational theories, and the related substitution facility by means of a rewrite system. There are some formal properties that the associated rewrite systems must satisfy when integrated into a computational framework. We have defined a comprehensive methodology for formalizing encodings in this way [8, 28].

The following examples show the formalization of some encodings appearing in the literature. This in order to show the applicability of the approach, and to see how the axiomatization can constitute a basis for investigating properties, advantages and disadvantages of a given naming device.

**Example 3.2**

Various encodings can be axiomatized by an equality theory: for example, a simple one where no information at all is included in any name. The encoding axiomatized by the following axiom corresponds to the non-ground encoding (identity function) typically used in Prolog meta-interpreters [63].

\[
\forall x \uparrow x = x. \tag{3.18}
\]

This encoding seems to have the advantage of simplicity, but, unfortunately, strongly reduces the expressive power of the metatheory. It is not possible, for example, to use a unification procedure for constructing names of expressions and accessing parts of them, as the name of the function symbol of a term is again a function symbol. A possible solution to this problem could be that of replacing axiom (3.18) above with the following two axioms.

\[
\text{For every constant } c, \quad \uparrow c = c. \tag{3.19}
\]

\[
\text{For every function symbol } f \text{ of arity } k, \quad \forall x_1 \ldots \forall x_k \uparrow (f(x_1, \ldots, x_k)) = \uparrow f \uparrow x_1 \ldots \uparrow x_k. \tag{3.20}
\]

In (3.20) the symbol $f$ appearing to the left of equality is a function symbol, while the $f$ appearing to the right of equality is a metaconstant. One advantage of using such overloading of names is that the rewrite system for such axioms can be very simple and efficient, but, on the other hand, ambiguous cases arise. Suppose, for example, that we want to find what the name term $[f, t_1, \ldots, t_k]$ names. Then we have an ambiguity because it could be either a name term of the form $[f, s_1, \ldots, s_k]$ or a term of the form $f(s_1, \ldots, s_k)$. (Jiang introduces an ambivalent logic [45] where he tackles this problem by making no distinction between sentences and terms.) For many metaprograms, however, such a representation is inadequate for other reasons: it does not allow us to investigate the instantiation of variables in queries. Actually, many kinds of metaprograms need to reason about the computational behaviour of the object program. In this case, a ground encoding appears to be more suitable.
The next example shows a simple form of ground encoding defined similarly to the Gödel numbering $\gamma$.

**Example 3.3**

Define first an *exponent* to be any natural number of the form $2^n$, for some $n \geq 0$, and an *assignment* to be any injective mapping from a finite subset of the set of variables and metavariables into the set of all exponents. We write assignments as $\{x_1/n_1, \ldots, x_k/n_k\}$, and by using this notation we assume that all variables and metavariables $x_i$ are distinct and all exponents $n_i$ are also distinct.

Let $t$ be a term and $x_1, \ldots, x_n$ be all variables and metavariables of $t$. Let $\theta$ be an assignment. The Gödel number $\gamma_\theta(t)$ of $t$ under $\theta$ is defined similarly to the Gödel numbering $\gamma$:

\[
\begin{align*}
\gamma_\theta(x_i) &= n_i \\
\gamma_\theta(c_i) &= 3^i \\
\gamma_\theta(f_j(t_1, \ldots, t_m)) &= 3^{j} \times 5^{\gamma_\theta(t_1)} \times \ldots \times p_{i+2}^{\gamma_\theta(t_m)},
\end{align*}
\]

where each $p_i$ is the $i$th prime number and the indexes of constants and function symbols are assumed to be distinct. We can formalize this encoding as follows.

\[
\uparrow 2^n = 2^n.
\]

For every constant $c_i$,

\[\uparrow c_i = 3^i.\]

For every function symbol $f_j$ of arity $k$,

\[
\forall x_1 \ldots \forall x_k \uparrow (f_j(x_1, \ldots, x_k)) = 3^{j} \times 5^{\gamma_\theta(x_1)} \times \ldots \times p_{k+2}^{\gamma_\theta(x_k)}.
\]

Then, given an assignment $\theta$, the ground representation of $t$ under $\theta$ is $\uparrow (t\theta)$.

Although simple and sound, the above encoding is inadequate for most knowledge-representation and computational purposes. A main property that a naming device should in fact in our opinion exhibit is compositionality: i.e. since a term is constructed (and deconstructed) by composing (decomposing) subterms, its name should correspondingly be constructed (and deconstructed) by composing (decomposing) names of subterms. The axioms below for the operators $\uparrow$ and $\downarrow$ are the basis of the formalization of the relationship between terms and the corresponding name terms. These axioms form a part of the equality theory for any ground encoding which is meant to be compositional. They just say that there exist names of names (each term can be referenced $n$ times, for any $n \geq 0$) and that the name of a compound term must be a function of the names of its components.

The axioms of the following equality theory, called $NT$ and first defined in [28], characterize name terms and compositional names for $HC^+$.  

**Definition 3.4**

Let $NT$ be the following equality theory.

- For every constant or metaconstant $c^n$, $n \geq 0$,
  \[\uparrow c^n = c^{n+1}.\]

- For every function symbol $f$ of arity $k$,
  \[\forall x_1 \ldots \forall x_k \uparrow (f(x_1, \ldots, x_k)) = [f^1, \uparrow x_1, \ldots, \uparrow x_k].\]

- For every compound name term $[X_0, X_1, \ldots, X_k]$ 
  \[\forall X_0 \ldots \forall X_k \uparrow [X_0, X_1, \ldots, X_k] = [\uparrow X_0, \uparrow X_1, \ldots, \uparrow X_k].\]
The simple examples above illustrate that an encoding directly determines the expressivity of the metatheory. If we consider an encoding that provides little information to the metalevel, then we can design efficient rewrite systems for that encoding; but, on the other hand, the expressivity of the metatheory is low (this is the case for an encoding along the lines of Example 3.2).

When an encoding has been established as being suitable for an application, its properties for sound and complete inference should be investigated. (In Section 5 we study what properties are required of a rewrite system for a suitable integration into a computational mechanism.) For example, encodings employing variable names result in a loss of completeness (see example below). However, such encodings allow the state of the computation (e.g. the instantiation of variables in queries) to be inspected. This capability is needed, for example, in applications that are mainly aimed at syntactic metaprogramming, like program manipulation and transformation via metaprograms. Thus, one may choose this last kind of encoding if these capabilities are important, provided that one is aware that certain other properties are lost.

**Example 3.5**
Consider any encoding providing names for variables and let \( P \) be the following definite program:

\[
p(x) \leftarrow Y = \uparrow x; q(Y) \quad \text{and} \quad q(a^1).
\]

The goal \( \leftarrow p(a) \) succeeds by first instantiating \( Y \) to \( a^1 \) and then proving \( \leftarrow q(a^1) \). In contrast, the goal \( \leftarrow p(x) \) fails, as \( Y \) is instantiated to the name of \( x \), say \( x^1 \), and the goal \( \leftarrow q(x^1) \) fails, \( x^1 \) and \( a^1 \) being distinct.

Furthermore, we observe that encodings influence the semantics of metalogic languages. In fact, metalanguages that are based on formally defined encodings have clear and well-defined declarative semantics. In contrast, giving a semantic account of a metalogic programming language that employs a trivial encoding is remarkably more difficult. This is easy to see for meta-interpreters, whose encoding mechanism has been outlined in Example 3.2. The difficulties associated with providing them with a reasonable semantics have been discussed in length by Barklund *et al.* [9].

The RCL system provides a default encoding which is compositional, and does not provide names for variables. The default encoding is in particular the one described in Definition 3.4. The system is, however, intended to be parametrical w.r.t. the naming device, i.e. the implementation can be adapted to the application domain at hand by replacing the default encoding with a new one. The new encoding should be defined along the lines given in this section, and implemented as specified in the next section (in most cases the new implementation will result in a modification of the existing one). Notice that we allow names, names of names, and so on. It is not easy to understand whether this could give problems such as circularity or unsoundness. This question is, however, solved by studying the properties of the rewrite system associated with the encoding, as illustrated in the following section.
3.4 An E-unification algorithm

In the context of equational logic programming, unification algorithms are usually expressed in terms of transformation systems based on sets of equations rather than on substitutions. In order to take into account names of the metalanguage \( HC^+ \), we define an E-unification algorithm based on a rewrite system for a given equality theory \( E \). To do this, we need some definitions (in the rest of the paper we will use the terminology of Dershowitz and Jouannaud [29]).

A rewrite rule over a set of terms is an ordered pair \( \langle l, r \rangle \) of terms, which we write as \( l \rightarrow r \).

The idea of rewriting is to impose directionality on the use of equations in proofs. A (finite) set \( R \) of rewrite rules is called a rewrite system.

We write the subterm of \( t \) rooted at position \( p \) as \( t[p] \). The term \( t \) with its subterm \( t[p] \) replaced by a term \( s \) is written as \( t[s]_p \). Given a rewrite system \( R \), a term \( s \) rewrites to a term \( t \), written as \( s \rightarrow t \), if \( s[p] = l \sigma \) and \( t = s[r] \), for some rule \( l \rightarrow r \) in \( R \), position \( p \) in \( s \), and substitution \( \sigma \). In that case, we say that \( s \) is reducible. A subterm \( s[p] \) at which a rewrite can take place is called redex; we say that \( s \) is in normal form if \( s \sigma \) has no redex for any substitution \( \sigma \). A term \( s \) is irreducible if it is not in normal form and it is not reducible.

That is, a term \( t \) is irreducible if \( t \) contains names that cannot be computed. For example, \( t \) might be \( \{x \} \) if the chosen encoding does not provide names for variables. A derivation in \( R \) is any (finite or infinite) sequence \( t_0 \rightarrow s_1 \rightarrow t_1 \rightarrow \cdots \) of applications of rewrite rules in \( R \). The derivability relation \( \frac{\Sigma}{R} \) is the reflexive, transitive closure of \( \rightarrow \). We write \( s \frac{1}{R} \rightarrow t \) if \( s \rightarrow n t \) and \( t \) is in normal form. We write the symmetric closure of \( \rightarrow \) as \( \leftrightarrow \). A rewrite system is terminating if there are no infinite derivations \( t_0 \rightarrow s_1 \rightarrow t_1 \rightarrow \cdots \) of terms. A rewrite system is convergent if all sequences of applications of rewrite rules lead to a unique normal form.

Given an equality theory \( E \), a rewrite system \( R \) is adequate for \( E \) if (i) \( R \) is terminating and (ii) \( s \frac{1}{R} \rightarrow t \) if and only if \( E \models s = t \). Hereafter, we write \( R_E \) to indicate any rewrite system adequate for \( E \).

A binding is an equation either of the form \( x = t \) if \( x \) is a variable that does not occur in the term \( t \), or of the form \( Y = s \) if \( Y \) is a metavariable that does not occur in the name term \( s \). A Herbrand assignment \( H = \{ x_1 = t_1, \ldots, x_k = t_k \} \) is a set of bindings such that the variables and metavariable \( x_i \) are pairwise distinct, no \( x_i \) is in any \( t_j \), and the terms \( t_1, \ldots, t_k \) are in normal form.

The intuition is that Herbrand assignments do not contain name equations, i.e. they do not contain equations with names that still have to be computed. This requirement allows us to have for each Herbrand assignment \( H \) an equivalent substitution \( \{ x_i / t_1, \ldots, x_k / t_k \} \), which we indicate with \( \hat{H} \).

A transformation rule, written as \( \Rightarrow \), is a rule that operates on triples of the form \( \langle H, F, S \rangle \), where \( H \) is a Herbrand assignment, \( F \) is a set of name equations and \( S \) is a set of equations. We can see \( H \) and \( F \) as the solved and unsolved part, and \( S \) as the set of equations still to be processed. \( H \) consists of all the bindings that have been computed, while \( F \) consists of name equations containing irreducible terms. A transformation system is a finite set of transformation rules. A transformation system is convergent if all sequences of transformations lead to a unique normal form.

Below we sketch a transformation system that extends Martelli and Montanari’s transfor-
Martelli and Montanari's transformation system is extended here with four new rules. The first new rule, \textit{swap variables}, is needed to swap the terms of an equation of the form \( y \overset{?}{=} x \) where \( y \) is a metavariable and \( x \) is a variable. By swapping those variables, we get an equation \( x \overset{?}{=} y \) that is a binding and can therefore be processed by \textit{eliminate}. The second, \textit{mutate}, allows us to compute names with respect to a given rewrite system \( R \). If a name equation \( e \) contains a redex \( c[p] \) reducible to \( t \), i.e., \( c[p] \overset{?}{=} t \), then \textit{mutate} replaces \( c[p] \) in \( e \) with \( t \). Finally, the rules \textit{freeze} and \textit{unfreeze} move name equations from \( S \) to \( F \), and vice versa. If a name equation \( x = t \) is irreducible, that is, \( t \) is not in normal form and contains names that cannot be computed, then \textit{freeze} moves \( x = t \) to the set \( F \). Such an equation remains in \( F \) until it becomes reducible, which is eventually allowed by means of a substitution applied to \( F \) by \textit{eliminate}. At this point, \textit{unfreeze} moves \( x = t \) back to \( S \), where it can subsequently be reduced.

\textbf{Definition 3.6}

Given a rewrite system \( R \), an \textit{E-unification algorithm}, written as \( \overset{\text{E}}{\Rightarrow} \), is any procedure that takes a finite set \( S_0 \) of equations, and uses the above transformation system to generate sequences of tuples from \( \langle \{\}, \{\}, S_0 \rangle \).

Starting with \( \langle \{\}, \{\}, S_0 \rangle \) and using the rules above until none is applicable results in \( \langle H, F, S \rangle \), where \( S \neq \{\} \), if and only if \( S_0 \) has no solution, or otherwise it results in a solved form \( \langle H, F, \{\} \rangle \), where \( H \) is a Herbrand assignment and \( F \) is a solvable set of irreducible name equations. Since the application of any of these rules preserves all solutions, the former situation corresponds to failure, while in the latter case a most general unifier can be extracted from \( H \). For the sake of simplicity, we have not specified the transformation rules needed to transform irreducible name equations in \( F \) to a solvable form. In any case, such a solvable form for \( F \) exists [28] which guarantees that \( \langle H, F, \{\} \rangle \) is solvable.
DEFINITION 3.7
Let $E$ be an equality theory. An $E$-solution of an equation $s = t$ is a Herbrand assignment $H$ such that $E \models sH = tH$. An $E$-solution of a set $S$ of equations is a Herbrand assignment that is an $E$-solution of every equation in $S$.

DEFINITION 3.8
Let $E$ be an equality theory. A system of rules is sound for $E$ if every rule in it preserves the set of all $E$-solutions.

The following four results are proved in [28].

PROPOSITION 3.9
Given an equality theory $E$ and a rewrite system $R_E$ adequate for $E$, the $E$-unification algorithm $\Rightarrow_E$ is sound for $E$ and terminating.

PROPOSITION 3.10
Let $R$ be a rewrite system. If $R$ is convergent, then the $E$-unification algorithm $\Rightarrow_R$ converges.

We present now a rewrite system based on the equality theory $NT$ of Definition 3.4. Recall that we write $c^n$ to indicate a constant $c$ named $n$ times; thus, $c$ may be written as $c^1$, its name as $c^1$, and so on.

DEFINITION 3.11
Let $UN$ be the following rewrite system. Let $\uparrow c^n \rightarrow c^{n+1}$
$\uparrow f(x_1, \ldots, x_n) \rightarrow [f^1, \uparrow x_1, \ldots, \uparrow x_n]$  
$\uparrow [X_0, \ldots, X_n] \rightarrow [\uparrow X_0, \ldots, \uparrow X_n]$  
$\uparrow \downarrow X \rightarrow X$  
$\downarrow c^{n+1} \rightarrow c^n$  
$\downarrow [f^1, X_1, \ldots, X_n] \rightarrow f(\downarrow X_1, \ldots, \downarrow X_n)$  
$\downarrow [f^{n+2}, X_1, \ldots, X_n] \rightarrow [\downarrow f^{n+2}, \downarrow X_1, \ldots, \downarrow X_n]$  
$\downarrow [X_0, X_1, \ldots, X_n] \rightarrow \downarrow [\downarrow X_0, \downarrow X_1, \ldots, \downarrow X_n]$  
$\downarrow \uparrow x \rightarrow x$

The rewrite system $UN$

EXAMPLE 3.12
With respect to $UN$, the $E$-unification algorithm rewrites

$$\langle \{\}, \{\}, \{f(X, Y, \uparrow X) \Rightarrow f(\uparrow a, \downarrow Z, Z)\} \rangle \Rightarrow_{UN} \langle \{X = a^1, Z = a^2, Y = a^1\}, \{\}, \{\} \rangle$$

in the following steps:
\[
\begin{align*}
\langle \{\}, \{f(X, Y, \top X) \equiv f(\top a, \downarrow Z, Z)\} \rangle &\Rightarrow \langle \{\}, \{X \equiv \top a, Y \equiv \downarrow Z, \top X \equiv Z\} \rangle \\
\langle \{\}, \{Y \equiv \downarrow Z\}, \{X \equiv \top a, \top X \equiv Z\} \rangle &\Rightarrow \langle \{\}, \{Y \equiv \downarrow Z, \{X \equiv \top a, \top X \equiv Z\}\} \rangle \\
\langle \{X = a^1\}, \{Y \equiv \downarrow Z\}, \{	op a^1 \equiv Z\}\rangle &\Rightarrow \langle \{X = a^1\}, \{Y \equiv \downarrow Z\}, \{a^2 \equiv Z\}\rangle \\
\langle \{X = a^1\}, \{Y \equiv \downarrow Z\}, \{Z \equiv a^2\}\rangle &\Rightarrow \langle \{X = a^1, Z = a^2\}, \{Y \equiv \downarrow a^2\}, \{\}\rangle \\
\langle \{X = a^1, Z = a^2, Y \equiv a^1\}\rangle &\Rightarrow \langle \{X = a^1, Z = a^2\}, \{\}, \{Y \equiv a^1\}\rangle \
\end{align*}
\]

**Proposition 3.13**
The rewrite system \(UN\) is adequate for \(NT\).

**Proposition 3.14**
The \(E\)-unification algorithm \(\Rightarrow\) is sound for \(NT\), terminates and converges.

### 3.5 \(E\)-interpretations

In this section we parametrize the semantics of the traditional Horn clause language w.r.t. an equality theory \(E\). To this aim the problem is that, whenever a semantics is defined over the Herbrand universe \(U\), equality is interpreted by default as syntactic identity. To overcome this restriction, Jaffar et al. [43] proposed the use of quotient universes. Here we adapt this technique to our context.

**Definition 3.15**
Let \(R\) be a congruence relation. The quotient universe of \(U\) with respect to \(R\), indicated as \(U/R\), is the set of the equivalence classes of \(U\) under \(R\), i.e. the partition given by \(R\) in \(U\).

Given an equality theory \(E\), there is an infinite number of models of \(E\). For \(E\) to have a canonical model, there must exist a congruence relation \(R\) such that

\[
E \models s = t \iff [s]_R = [t]_R
\]

where \([s]_R\) and \([t]_R\) denote the \(R\)-equivalence classes of the ground term \(s\) and \(t\), i.e. \([s]_R = \{x \mid x R s\}\). This can be achieved only if the equality theory has a ‘finest’ congruence relation (in the sense of set inclusion). Jaffar et al. showed that each consistent (Horn clause) equality theory generates a finest congruence relation \(R_0\) (the intersection of all congruence relations that are models of \(E\)). As a consequence, it holds that

\[
(P, E) \models A \iff P \models_{U/R_0} A
\]

where \((P, E)\) is a logic program, \(A\) is a ground atom and \(\models_{U/R_0}\) denotes logical implication in the context of \(U/R_0\). Thus we can work in a fixed domain which is the canonical domain for \((P, E)\).

In the following, we write \(U/E\) for \(U/R_0\), \([s]\) for the element in \(U/E\) assigned to the ground term \(s\) and, for any predicate symbol \(p\), we write \([p(t_1, \ldots, t_n)]\) as a shorthand for \(p([t_1], \ldots, [t_n])\).

We can now introduce the definitions of \(E\)-base, \(E\)-interpretation and \(E\)-model of a logic program \((P, E)\).
Definition 3.16
The E-base $B_{(P,E)}$ of a logic program $(P,E)$ is the set of all atoms which can be formed by using predicate symbols from the language of $(P,E)$ with elements from the quotient universe $U/E$ as arguments.

Definition 3.17
An E-interpretation of a logic program $(P,E)$ is any subset of $B_{(P,E)}$.

Definition 3.18
Let $I$ be an E-interpretation. Then $I$ E-satisfies a ground definite clause $A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$ if and only if at least one of the following conditions hold:

1. $E \not\models e_i$, for some $i, 1 \leq i \leq q$.
2. $[A_j] \not\in I$, for some $j, 1 \leq j \leq m$, or
3. $[A] \in I$.

Definition 3.19
Let $I$ be an E-interpretation of a logic program $(P,E)$. Then $I$ E-satisfies $(P,E)$ if and only if $I$ E-satisfies each ground instance of every clause in $P$. If there exists an E-interpretation $I$ which E-satisfies $(P,E)$, then $(P,E)$ is E-satisfiable, otherwise $(P,E)$ is E-unsatisfiable.

Definition 3.20
Let $I$ be an E-interpretation of a logic program $(P,E)$. Then $I$ is an E-model of $(P,E)$ if and only if $I$ E-satisfies $(P,E)$.

Definition 3.21
A ground atom $A$ is a logical E-consequence of a logic program $(P,E)$ if, for every E-interpretation $I$, $I$ is an E-model of $(P,E)$ implies that $[A] \in I$.

The least E-model of a logic program $(P,E)$ can be characterized as the least fixed point of a mapping $T_{(P,E)}$ over E-interpretations [43], written as lfp$(T_{(P,E)})$. Let ground$(P)$ be the set of all ground instances of clauses in $P$.

Definition 3.22
Let $I$ be an E-interpretation of a logic program $(P,E)$. Then $T_{(P,E)}$ is defined as follows:

$$T_{(P,E)}(I) = \{ [A] : (A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m) \in \text{ground}(P),$$

$$E \models e_i \text{ for } 1 \leq i \leq q,$$

$$[A_j] \in I \text{ for } 1 \leq j \leq m \}.$$

The following result is proved by Jaffar et al. [43].

Theorem 3.23
$M_{(P,E)} = \text{lfp}(T_{(P,E)}) = T_{(P,E)} \uparrow \omega$.

In summary, we have defined an enhanced Horn clause language $HC^+$ which allows users to introduce their own naming convention by means of an equality theory $E$. The semantics of $HC^+$ is, up to now, just the semantics of the traditional Horn clause language, which has been made parametrical w.r.t. $E$ by means of the technique of quotient universes. This is the first step of the definition of $RCL$, in which we have provided users with a language powerful enough to represent knowledge and metaknowledge in a deductive system $DS$.

Then, we have to provide the possibility of defining the inference rules of $DS$ and performing deductions in $DS$. With this aim, in Section 2 we have introduced a formal device,
that we have called reflection, for defining new inference rules and integrating them into SLD-resolution. The novelty of the approach is precisely that newly defined inference rules are immediately ‘executable’, in the context of a declarative and procedural semantics which do not depart from the usual ones. In fact, the following sections give a model-theoretic and functional characterization of logic programs with naming and reflection, and present an extension to SLD-resolution that takes reflection principles into consideration.

4 Reflective semantics

4.1 Reflective $E$-models and fixed point semantics

We use the following definitions.

**Definition 4.1**
Let $R$ be a reflection principle and $I$ an $E$-interpretation of a logic program $(P, E)$. Then $I$ reflectively $E$-satisfies $(P, E)$ (with respect to $R$) if and only if $I$ $E$-satisfies $(P \cup R(P), E)$.

**Definition 4.2**
If there exists an $E$-interpretation $I$ that reflectively $E$-satisfies a logic program $(P, E)$, then $(P, E)$ is reflectively $E$-satisfiable, otherwise $(P, E)$ is reflectively $E$-unsatisfiable.

**Definition 4.3**
Let $I$ be an $E$-interpretation of a logic program $(P, E)$. Then $I$ is a reflective $E$-model of $(P, E)$ if and only if $I$ reflectively $E$-satisfies $(P, E)$.

Reflective $E$-models are clearly models in the usual sense [44], as they are obtained by extending a given logic program $(P, E)$ with a set of definite clauses. Therefore the model intersection property still holds and there exists a least reflective $E$-model of $(P, E)$, indicated as $M^R_{\langle P, E \rangle}$. It entails the consequences of $(P, E)$, the additional consequences drawn by means of the reflection axioms, and the further consequences obtained from both. $M^R_{\langle P, E \rangle}$ is in general not minimal as an $E$-model of $(P, E)$, but it is minimal with respect to the set of consequences which can be drawn from both the logic program and the reflection axioms.

**Definition 4.4**
A ground atom $A$ is a reflective logical $E$-consequence of a logic program $(P, E)$ if, for every $E$-interpretation $I$, $I$ is a reflective $E$-model for $(P, E)$ implies that $[A] \in I$.

Given a logic program $(P, E)$ and a definite goal $G$, we hereafter write $(P, E) \cup \{G\}$ for $(P \cup \{G\}, E)$ to enhance readability.

**Proposition 4.5**
Let $(P, E)$ be a logic program and $\leftarrow A_1, \ldots, A_k$ a ground definite goal. Then $(P, E) \cup \{\leftarrow A_1, \ldots, A_k\}$ is reflectively $E$-unsatisfiable if and only if $A_1 \land \ldots \land A_k$ is a reflective logical $E$-consequence of $(P, E)$.

The least reflective $E$-model of a logic program $(P, E)$ can be characterized as the least fixed point of a mapping $T^R_{\langle P, E \rangle}$ that extends $T_{\langle P, E \rangle}$ [43]. The extension is based on the presence of reflection axioms.

**Definition 4.6**
Let $R$ be a reflection principle and $I$ an $E$-interpretation of a logic program $(P, E)$. $T^R_{\langle P, E \rangle}$ is defined as follows:
Next we give a fixed point characterization of the least reflective \( E \)-model of a logic program.

**Proposition 4.7**

Let \( \mathcal{R} \) be a reflection principle and \( I \) an \( E \)-interpretation of a logic program \((P, E)\). The mapping \( T_{(P, E)}^{\mathcal{R}} \) is continuous.

The class of reflective \( E \)-models can be characterized in terms of \( T_{(P, E)}^{\mathcal{R}} \).

**Proposition 4.8**

Let \( \mathcal{R} \) be a reflection principle and \( I \) an \( E \)-interpretation of a logic program \((P, E)\). \( I \) is a reflective \( E \)-model of \((P, E)\) if and only if \( T_{(P, E)}^{\mathcal{R}}(I) \subseteq I \).

As the class of \( E \)-interpretations forms a complete lattice under the inclusion order [43], \( T_{(P, E)}^{\mathcal{R}} \) is continuous over this class, and the class of reflective \( E \)-models is given by \( \{ I \mid T_{(P, E)}^{\mathcal{R}}(I) \subseteq I \} \). The result of Jaffar et al. [43] is thus applicable, and provides a fixed point characterization of the least reflective \( E \)-model of a logic program \((P, E)\).

**Theorem 4.9**

Let \( \mathcal{R} \) be a reflection principle and \((P, E)\) a logic program. Then, \( M_{(P, E)}^{\mathcal{R}} = \text{lfp}(T_{(P, E)}^{\mathcal{R}}) = T_{(P, E)}^{\mathcal{R}} \uparrow \omega \).

Next we introduce the definitions of answer and correct answer.

**Definition 4.10**

Let \((P, E)\) be a logic program and \( G \) a definite goal. An answer for \((P, E) \cup \{ G \}\) is a pair \( \langle H, F \rangle \) consisting of a Herbrand assignment \( H \) and a set \( F \) of irreducible name equations.

**Definition 4.11**

Let \((P, E)\) be a logic program, \( G \) a definite goal \( \leftarrow A_1, \ldots, A_k \), and \( \langle H, F \rangle \) an answer for \((P, E) \cup \{ G \}\). \( \langle H, F \rangle \) is a correct answer for \((P, E) \cup \{ G \}\) if, for every \( E \)-solution \( H' \) of \( F, \forall((A_1 \land \ldots \land A_k)\widehat{H} \widehat{H'}) \) is a reflective logical \( E \)-consequence of \((P, E)\).

**Theorem 4.12**

Let \((P, E)\) be a logic program and \( \leftarrow A_1, \ldots, A_k \) a definite goal. Suppose that \( \langle H, F \rangle \) is an answer for \((P, E) \cup \{ \leftarrow A_1, \ldots, A_k \}\) and \( H' \) is an \( E \)-solution of \( F \). If \((A_1 \land \ldots \land A_k)\widehat{H} \widehat{H'}\) is ground, then the following are equivalent:

(a) \( \langle H, F \rangle \) is a correct answer.
(b) \( (A_1 \land \ldots \land A_k)\widehat{H} \widehat{H'} \) is true w.r.t. every reflective \( E \)-model of \((P, E)\).
(c) \( (A_1 \land \ldots \land A_k)\widehat{H} \widehat{H'} \) is true w.r.t. the least reflective \( E \)-model of \((P, E)\).

### 4.2 SLD\(^{\mathcal{R}}\)-resolution

It is well known how to reformulate SLD-resolution over definite programs in terms of sets of equations rather than substitutions (see Clark [17]). A computation state is a pair \( \langle M, H \rangle \),
where $M$ is a set of atoms that have to be proved and $H$ is a Herbrand assignment. Unification can in this process be seen as a rewrite system that takes a set of equations to an equivalent Herbrand assignment [52].

The assumption that unification rewrites the whole set of equations to a Herbrand assignment can be relaxed. Let a state instead consist of a triple $\langle M, H, F \rangle$, where $M$ is a set of atoms, $H$ is a Herbrand assignment, and $F$ is a set of irreducible name equations. We can see $H$ and $F$ as the solved and unsolved part of a single equation system.

Given $n$ equations $e_1, \ldots, e_n$, an equality theory $E$ and a rewrite system $R$ adequate for $E$, a transformation system for unification takes a triple $\langle H, F, \{e_1, \ldots, e_n\} \rangle$ either to a triple $\langle H', F', S' \rangle$, where $S' \neq \{\}$, if $\{e_1, \ldots, e_n\}$ is not solvable, or to a solved form $\langle H', F', \{\} \rangle$ such that $H \subseteq H'$ and $H \cup F \cup \{e_1, \ldots, e_n\}$ is equivalent to $H' \cup F'$ under $E$.

Let $(P, E)$ be a logic program and $G$ a definite goal $\leftarrow B_1, \ldots, B_r$. An initial state for refuting $(P, E) \cup \{G\}$ is a triple $\langle \{B_1, \ldots, B_r\}, \{\} \rangle$ and a success state is a triple $\langle \{\}, H, F \rangle$, where $F$ is a solvable set of irreducible name equations, i.e. there exists a Herbrand assignment $H'$ such that $E \models F \hat{H}'$.

Now we can extend SLD-resolution to take into consideration a reflection principle $\mathcal{R}$. We call the extended SLD-resolution SLD$^\mathcal{R}$-resolution.

Given a reflection principle $\mathcal{R}$ and an equality theory $E$, we write $\Delta_{\mathcal{R}}$ to indicate any procedure that computes $\mathcal{R}$, and $R_E$ to indicate any rewrite system adequate for $E$.

**DEFINITION 4.13**

Let $\mathcal{R}$ be a reflection principle and $(P, E)$ a logic program. Let $\langle M \cup \{p(t_1, \ldots, t_n)\}, H, F \rangle$ be a state. Given a variant $C$ of a definite clause in $P$, the state $\langle M \cup \{A_1, \ldots, A_m\}, H', F' \rangle$ is derived from $\langle M \cup \{p(t_1, \ldots, t_n)\}, H, F \rangle$ and $C$ by using $\Delta_{\mathcal{R}}$ and $R_E$ if either

(a) $C$ is $p(t_1, \ldots, t_n) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$ or

(b) $(p(t_1, \ldots, t_n) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m) \in \Delta_{\mathcal{R}}(C),$

and $\langle H, F, \{t_1 = t'_1, \ldots, t_n = t'_n, e_1, \ldots, e_q\} \rangle \Rightarrow_{n_{\mathcal{R}}} \langle H', F', \{\} \rangle$.

The first case (a) corresponds to the operations of the modified SLD-resolution discussed above. The second case (b) is based on the use of reflection axioms obtained by means of $\Delta_{\mathcal{R}}$.

The additional inference rule could also be added to other inference systems for definite programs that have provisions for delaying computation.

An SLD$^\mathcal{R}$-derivation is a (finite or infinite) path in the tree of states above. An SLD$^\mathcal{R}$-refutation is a finite path in the tree ending with a success state.

**DEFINITION 4.14**

Let $\mathcal{R}$ be a reflection principle, $(P, E)$ a logic program and $G$ a definite goal. An SLD$^\mathcal{R}$-derivation of $(P, E) \cup \{G\}$ consists of a (finite or infinite) sequence of states $\langle M, \{\} \rangle$, $\langle M_1, H_1, F_1 \rangle$, ..., and a sequence $C_1, C_2, \ldots$ of variants of definite clauses of $P$, such that each $\langle M_{i+1}, H_{i+1}, F_{i+1} \rangle$ is derived from $\langle M_i, H_i, F_i \rangle$ and $C_{i+1}$ by using $\Delta_{\mathcal{R}}$ and $R_E$.

**DEFINITION 4.15**

Let $\mathcal{R}$ be a reflection principle, $(P, E)$ a logic program and $G$ a definite goal. An SLD$^\mathcal{R}$-refutation of $(P, E) \cup \{G\}$ is a finite SLD$^\mathcal{R}$-derivation of $(P, E) \cup \{G\}$ which has a success state as last state in the derivation. If the success state is of the form $\langle \{\}, H_n, F_n \rangle$, we say that the refutation has length $n$. 

5 Properties of SLD\textsuperscript{R} -resolution

In this section we present the results of soundness and completeness of SLD\textsuperscript{R} -resolution with respect to the least reflective \( E \)-model.

5.1 Soundness

To prove soundness of SLD\textsuperscript{R} -resolution we use the following definition.

DEFINITION 5.1
Let \((P, E)\) be a logic program and \(G\) a definite goal. Suppose that \(\{\{\}, H, F\}\) is the success state of an SLD\textsuperscript{R} -refutation of \((P, E) \cup \{G\}\). Then \(\{H, F\}\) is a computed answer for \((P, E) \cup \{G\}\).

The next theorem states the main soundness result, i.e. that computed answers are correct.

THEOREM 5.2
(Soundness of SLD\textsuperscript{R} -resolution) Let \((P, E)\) be a logic program and \(G\) a definite goal. Every computed answer for \((P, E) \cup \{G\}\) is a correct answer for \((P, E) \cup \{G\}\).

Furthermore, the following result is an immediate consequence.

COROLLARY 5.3
Let \((P, E)\) be a logic program and \(G\) a definite goal. Suppose that there exists an SLD\textsuperscript{R} -refutation of \((P, E) \cup \{G\}\). Then \((P, E) \cup \{G\}\) is reflectively \( E \)-unsatisfiable.

DEFINITION 5.4
The success set of a logic program \((P, E)\) is the set of all ground atoms \(A\) such that \((P, E) \cup \{\leftarrow A\}\) has an SLD\textsuperscript{R} -refutation.

Notice that atoms in the success set need not be in normal form, that is, they may contain occurrences of the operators \(\uparrow\) and \(\downarrow\).

As ground atoms may contain occurrences of \(\uparrow\) and \(\downarrow\), while reflective \( E \)-models only contain representative forms of such atoms, the success set of a logic program is in general not contained in its least reflective \( E \)-model. However, this property holds if we consider the representative forms of ground atoms. (Recall that the representative form of a ground atom \(A\) is written as \([A]\).)

COROLLARY 5.5
If a ground atom \(A\) belongs to the success set of a logic program \((P, E)\), then \([A]\) is contained in the least reflective \( E \)-model of \((P, E)\).

Now we strengthen Corollary 5.5 by showing that, if a ground atom \(A\) has an SLD\textsuperscript{R} -refutation of length \(n\), then \([A]\) \( \in T_{(P, E)}^{\rightarrow} \uparrow n\). This is an extension of the result due to Apt and van Emden [4]. We use the following definition.

DEFINITION 5.6
The closure of an atom \(A\), indicated as \(\Psi(A)\), is the set of representative elements of all ground instances of \(A\).

\[\Psi(A) = \{ [B] \mid \text{for every ground instance } B \text{ of } A \}.\]

THEOREM 5.7
Let \((P, E)\) be a logic program and \(G\) a definite goal \(\leftarrow A_1, \ldots, A_k\). Suppose \((P, E) \cup \{G\}\) has an SLD\textsuperscript{R} -refutation of length \(n\) with computed answer \((H_n, F_n)\).

Then, \(\bigcup_{j=1}^{k} \Psi(A_j H_n H') \subseteq T_{(P, E)}^{\rightarrow} \uparrow n\), for every \( E \)-solution \(H'\) of \(F_n\).
5.2 Completeness

The main result of this section is the completeness of SLD\(^R\)-resolution. This result holds if the transformation system that is the parameter of SLD\(^R\)-resolution converges.

We begin our argument for completeness by appropriately rephrasing the Lifting lemma [49]. This lemma essentially states that, if we can prove a goal \(G \land H\) from a logic program, then we can also prove the less instantiated goal \(G\). The two proofs have the same length and are such that the computed answer of \(G \land H\) can be obtained from the one of \(G\) by taking into consideration the bindings contained in \(H\).

**Lemma 5.8**
(Lifting lemma) Let \((P, E)\) be a logic program, \(H\) a Herbrand assignment and \(G\) a definite goal. Suppose there exists an SLD\(^R\)-refutation of \((P, E) \cup \{G \land H\}\) with success state \(\langle \{\}, H_n, F_n \rangle\). If \(R_E\) is convergent, then there exists an SLD\(^R\)-refutation of \((P, E) \cup \{G\}\) of the same length with success state \(\langle \{\}, H'_n, F'_n \rangle\) such that \(\langle H'_n, F'_n, H\rangle \implies _{n_E} \langle H_n, F_n, \{\}\rangle\).

The first completeness result gives the converse of Corollary 5.5.

**Theorem 5.9**
Let \((P, E)\) be a logic program. A ground atom \(A\) belongs to the success set of \((P, E)\) if and only if \([A]\) is contained in the least reflective \(E\)-model of \((P, E)\).

**Theorem 5.10**
Let \((P, E)\) be a logic program and \(G\) a definite goal. Suppose that \((P, E) \cup \{G\}\) is reflectively \(E\)-unsatisfiable. Then there exists an SLD\(^R\)-refutation of \((P, E) \cup \{G\}\).

Next we turn attention to correct answers. It is not possible to prove the exact converse of Theorem 5.2 because computed answers are always more 'general' than correct answers with respect to the variables and the metavariables \(x_1, \ldots, x_n\) contained in the definite goal. However, we can prove that every correct answer is an instance of a computed answer with respect to \(x_1, \ldots, x_n\). To do this, we use the following result.

**Lemma 5.11**
Let \((P, E)\) be a logic program and \(A\) an atom. Suppose that \(x_1, \ldots, x_n\) are all the variables and the metavariables occurring in \(A\) and that \(\forall x_1 \ldots \forall x_n. A\) is a reflective logical \(E\)-consequence of \((P, E)\). Then, there exists an SLD\(^R\)-refutation of \((P, E) \cup \{\leftarrow A\}\) with computed answer \(\langle H, F\rangle\) such that \(E \models \forall x_1 \ldots \forall x_n. (H \cup F)\).

**Example 5.12**
Consider the equality theory \(NT\). Let \((P, E)\) be the logic program:

\[
(\{p(x) \leftarrow Y = \uparrow x\}, NT),
\]

where \(x\) is a variable and \(Y\) a metavariable. Then, \(\forall z. (p(z))\) is a reflective logical \(E\)-consequence of \((P, E)\). In fact, the name equation \(Y = \uparrow x\) is satisfied for every value of \(x\). A computed answer for the goal \(\leftarrow p(z)\) is \(\langle \{z = x\}, \{Y = \uparrow x\}\rangle\). It holds that \(UN \models \forall y. \exists x. \exists y. (z = x \land Y = \uparrow x)\).

Now we are in the position to state the main completeness result.

**Theorem 5.13**
(Completeness of SLD\(^R\)-resolution) Let \((P, E)\) be a logic program and \(G\) a definite goal. If \(R_E\) is convergent, then for every correct answer \(\langle H, F\rangle\) for \((P, E) \cup \{G\}\), there exists a computed answer \(\langle H', F'\rangle\) for \((P, E) \cup \{G\}\). Furthermore, there exists a Herbrand
assignment $H''$ such that, for every $E$-solution $H_F$ and $H_{F'}$ of $F$ and $F'$, respectively, $(G \hat{H} \hat{H}_F) \hat{H}'' = G \hat{H} \hat{H}_F$ holds.

6 Applications

The main proposal of the present paper is the novel use of reflection principles as a paradigm for the representation of knowledge in a computational logic setting. The claim is that in many cases well-chosen reflection principles can adequately, clearly and concisely represent the basic features and properties of a domain. Though some technical developments shown in this paper are quite intricate, they serve as the behind-the-scenes sound definition and operation of the proposed system. Users will not be concerned with most of them, except for those which are aimed at helping users to tailor the system to their specific needs.

To substantiate this claim, in this section we offer examples of how to use the system capabilities in three different representation problems. Overall, we hope that this section also shows how one concept and tool (i.e. reflection principles) can be used in such different application areas, that they would otherwise be (and in the literature are) handled by different formalisms and techniques; in other words, reflection principles actually work as a knowledge representation paradigm.

On purpose, we recall some of the application domains we have studied in the past, so as to show how the previous ad hoc formulations can be rephrased as particular instances of the new framework that we propose in this paper.

6.1 Reflective Prolog

The first example of application of RCL to the definition of an actual deductive system concerns a metalogic Horn clause language with an extended resolution principle. This language is called Reflective Prolog, and is described in detail in [24]. Reflective Prolog (RP for short) has been defined and implemented by (some of) the authors of this papers: for them it has been the seminal work which stimulated the first intuition of the concepts that, with time and thought, have led to the formalization of RCL. Then, turning back, it is interesting to see how the new general framework we are now presenting is able to express that language that is, in a sense, its ancestor.

The axiomatization of the naming mechanism of Reflective Prolog as an equality theory (computationally characterized by a rewrite system), which is the first step for defining Reflective Prolog in RCL, is described in [9].

As concerns the Reflective Prolog inference rule, i.e. RSLD-resolution, we may notice that it can be seen as a form of SLD$^R$-resolution which uses reflection axioms implicitly present in the program. Thus, RSLD-resolution can be expressed in RCL by two reflection principles: reflection down and reflection up.

Reflection down makes any conclusion drawn at the metaevaluation level available (reflected down) to the object level. Reflection down can be represented by the following reflection principle $\mathcal{D}$. Let $C$ be a definite clause.

- If $C$ is of the form $\text{solve}([p^1, t_1, \ldots, t_n]) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$, then

$$\mathcal{D}(C) = \{ p(x_1, \ldots, x_n) \leftarrow x_1 = \downarrow t_1, \ldots, x_n = \downarrow t_n, e_1, \ldots, e_q, A_1, \ldots, A_m \}.$$
Reflection Principles in Computational Logic

Let 

\[ D(C) = \begin{cases} 
  p(x_1, \ldots, x_n) \leftarrow x_1 = \downarrow t_1, \ldots, x_n = \downarrow t_n, \\
  X = p^1, \\
  e_1, \ldots, e_q, A_1, \ldots, A_m & \text{for every predicate symbol } p \neq \text{solve} 
\end{cases} \]

If \( C \) is of the form \( \text{solve}(X) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m \), then

\[ D(C) = \begin{cases} 
  p(y_1, \ldots, y_n) \leftarrow y_1 = \downarrow X_1, \ldots, y_n = \downarrow X_n, \\
  X = [p^1, X_1, \ldots, X_n], \\
  e_1, \ldots, e_q, A_1, \ldots, A_m & \text{for every predicate symbol } p \neq \text{solve} 
\end{cases} \]

Reflection up makes any conclusion drawn at the object level available (reflected up) to the metaevaluation level. Reflection up can be represented by the following reflection principle called \( \mathcal{U} \).

\[ \mathcal{U}(C) = \{ \text{solve}([p^1, X_1, \ldots, X_n]) \leftarrow X_1 = \uparrow t_1, \ldots, X_n = \uparrow t_n, e_1, \ldots, e_q, A_1, \ldots, A_m \}. \]

RSLD-resolution can then be defined by the following reflection principle \( \mathcal{R} \).

\[ \mathcal{R}(C) = \begin{cases} 
  \mathcal{U}(C) & \text{if } C \text{ is an object level clause} \\
  D(C) & \text{if } C \text{ is a metaevaluation clause} 
\end{cases} \]

Thus, SLD\(^{\mathcal{R}} \) -resolution is able to use clauses with conclusion \( \text{solve}(X) \) to resolve a goal \( A \) (downward reflection), and, vice versa, clauses with conclusion \( A \) to resolve a goal \( \text{solve}(X) \) (upward reflection).

Below we reformulate in RCL an old example, which shows how metaevaluation clauses can play the role of additional clauses for object level predicates.

Example 6.1
Let \( (P, E) \) be the following logic program:

\[ \left\{ \begin{array}{l}
  \text{solve}([X, Y, Z]) \leftarrow \text{symmetric}(X), \text{solve}([X, Z, Y]) \\
  \text{symmetric}(p^1) \\
  p(a, b)
\end{array} \right\}, NT \]

where \( NT \) is the equality theory defined in Definition 3.4. The first clause in \( P \) defines the usual concept of symmetry of a relation: the objects with names \( Y \) and \( Z \) are in the relation with name \( X \), provided that the relation denoted by \( X \) is asserted to be symmetric and that the objects denoted by \( Z \) and \( Y \) are in the relation denoted by \( X \). The second clause states that the relation \( p \) is symmetric, and the last clause partially defines the relation \( p \).

As \( p \) is the only binary predicate symbol in \( P \), the reflection axioms of \( P \) are the following:

\[ \mathcal{R}(P) = \begin{cases} 
  p(y_1, z_1) \leftarrow y_1 = \downarrow Y, z_1 = \downarrow Z, X = p^1, \\
  \text{symmetric}(X), \text{solve}([X, Z, Y]) \\
  \text{solve}([\text{symmetric}^1, X]) \leftarrow X = \uparrow p^1 \\
  \text{solve}([p^1, X, Y]) \leftarrow X = \uparrow a, Y = \uparrow b 
\end{cases} \]
Now we can prove \( p(b, a) \) from \( P \) by applying SLD\(^{RP}\)-resolution.

Notice that \( p(b, a) \) does not logically follow from \((P, E)\) without reflection principles. In fact, the least \( E\)-model and reflective \( E\)-model of \((P, E)\) are respectively:

\[
M_{(P, E)} = \{ [p(a, b)], [\text{symmetric}(p^1)] \} \\
M^{RP}_{(P, E)} = M_{(P, E)} \cup \{ [p(b, a)], [\text{solve}([p^1, a^1, b^1])], [\text{solve}([\text{symmetric}^1, p^2])], [\text{solve}([p^1, b^1, a^1])] \}.
\]

Thus, by means of reflection up and reflection down, the first clause of \( P \) becomes an axiomatization of symmetry, which can be applied whenever necessary.

In summary, it can be useful to explicitly state the difference between Reflective Prolog as it was originally defined, and its formalization in RCL.

- RP had an ad hoc extended unification treating a fixed naming, while in RCL the naming is axiomatized and treated by means of rewrite rules.
- RP had a unique hard-wired reflection principle, while in RCL any reflection principle most appropriate to the domain can be expressed; this also implies that the above reflection principle could coexist in the same system with other reflection principles, for instance those introduced in the following subsections.
- RP semantics was defined in a specific way, while its reformulation in RCL is given a semantics as an instance of the general schema given in previous sections. Precisely, the concepts of extended Herbrand base and extended interpretation were absolutely ad hoc; the concept of a reflective model for RP can be considered as a rough first sketch which, in time, has evolved into the more general concept presented in this paper.

### 6.2 Communication-based reasoning

Another problem that we have discussed in a previous paper [20] concerns the ability to represent agents and multi-agent cooperation, which is central to many AI applications. In the context of communication-based reasoning, the interaction among agents is based on communication acts.

**Communication acts** are formalized by means of the predicate symbols \( \text{tell} \) and \( \text{told} \). They both take as first argument the name of a theory symbol and as second argument the name of an expression of the language. Let \( \omega \) and \( \phi \) be theory symbols and \( A \) an atom. The intended meaning of \( \omega: \text{tell}(\phi^1, A^1) \) is: the agent \( \omega \) tells agent \( \phi \) that \( A \), and of \( \phi: \text{told}(\omega^1, A^1) \) is: \( \phi \) is told by \( \omega \) that \( A \). These two predicates are intended to model the simplest and most neutral form of communication among agents, with no implication about provability (or truth, or whatever) of what is communicated, and no commitment about how much of its information an agent communicates and to whom.

The intended connection between \( \text{tell} \) and \( \text{told} \) is formalized by the following reflection principle \( C \).

- If \( C \) is a clause of the form \( \omega: \text{tell}(\phi^1, Z) \leftarrow \omega e_1, \ldots, \omega e_\phi, \omega: B_1, \ldots, \omega: B_n \), then

\[
C(C) = \{ x: \text{told}(Y, Z) \leftarrow \omega (x = \downarrow \phi^1), x: (Y = \uparrow \omega), \\
\omega e_1, \ldots, \omega e_\phi, \omega: B_1, \ldots, \omega: B_n \}.
\]
Its intuitive meaning is that every time an atom of the form \( \text{tell}(\phi^1, Z) \) can be derived from a theory \( \omega \) (which means that agent \( \omega \) wants to communicate proposition \( Z \) to agent \( \phi \)), the atom \( \text{told}(Y, Z) \) is consequently derived also in the theory \( \phi \) (which means that proposition \( Z \) becomes available to agent \( \phi \)).

We propose an example to show in some detail what is the declarative semantics of a program, and how SLD\(^\omega \)-resolution works.

**Example 6.2**

Consider the equality theory \( NT \). Let \((P, E)\) be the logic program:

\[
\left\{ \begin{array}{l}
\omega:\text{tell}(\phi^1, \text{ciao}^1) \leftarrow \omega:\text{friend}(\phi^1) \\
\omega:\text{friend}(\phi^1) \\
\phi:\text{hate}(\omega^1)
\end{array} \right\}, \ NT
\]

The reflection axioms of \( P \) are the following:

\[
C(P) = \left\{ x:\text{told}(Y, \text{ciao}^1) \leftarrow \omega : (x = \downarrow \phi^1), x : (Y = \uparrow \omega), \omega:\text{friend}(\phi^1) \right\}.
\]

The least \( E \)-model and reflective \( E \)-model of \((P, E)\) are respectively:

\[
M_{(P, E)} = \{ [\omega : \text{friend}(\phi^1)], [\phi : \text{hate}(\omega^1)], [\omega : \text{tell}(\phi^1, \text{ciao}^1)] \}
\]

\[
M^E_{(P, E)} = M_{(P, E)} \cup \{ [\phi : \text{told}(\omega^1, \text{ciao}^1)] \}.
\]

The goal \( \leftarrow \phi : \text{told}(\omega^1, Z) \) can be proved with the following steps:

- The initial state is \( \{ \{ \phi : \text{told}(\omega^1, Z) \}, \{ \}, \{ \} \} \).
- By applying the second case (b) of Definition 4.13 with respect to \( C \), we obtain the state \( \langle \{ \omega : \text{friend}(\phi^1) \}, \{ x = \phi, Y = \omega^1, Z = \text{ciao}^1 \}, \{ \} \rangle \).
- Finally, by considering the second clause in \( P \), we obtain the final state \( \langle \{ \}, \{ x = \phi, Y = \omega^1, Z = \text{ciao}^1 \}, \{ \} \rangle \).

### 6.3 Plausible reasoning

Plausible reasoning is a suitable realm of application of reflection principles. In fact, most forms of plausible reasoning reinterpret available premisses to draw plausible conclusions.

In logic programming, given a program \( P \), viewed as divided into two subprograms \( P_s \) and \( P_t \) (which play the role of the source and the target domain, respectively), analogy can be procedurally performed by transforming rules in \( P_s \) into analogous rules in \( P_t \). The analogous rules can be computed by means of *partial identity* between terms of the two domains [37], or by means of *predicate analogies* and *term correspondence* [25].

In particular, let us assume that predicates with the same name in \( P_s \) and \( P_t \) are in analogy by default. Let us also assume an explicit declaration is provided of analogy between predicates or, more generally, between terms of the two programs (this declaration is called *term correspondence*). Then, given a goal which is not provable in \( P_t \), this goal may possibly be provable by analogy, and in particular by adapting a suitably selected rule of the source program \( P_s \). Given a term correspondence, this rule can be transformed into an analogous rule, composed of predicates and terms of the target program, to be used in proving the given
Reflection Principles in Computational Logic

A deductive system which acts in this way can be easily formalized in RCL.

In this case no encoding device is needed (this is not a metaprogramming application). Nevertheless, the machinery for defining encodings can be ‘recycled’ for defining term analogies. In particular, substitutions used for unification can be seen as particular cases of correspondences. Thus, term correspondences can be composed with substitutions, giving a new term correspondence as a result.

The inference rule implementing this kind of analogical reasoning can be expressed in terms of a reflection principle defined below. Given a set $S$ of predicate analogies and a term correspondence $\sigma$, define a relation $r$ as:

1. $r(p(t_1, \ldots, t_n), q(t_1\sigma, \ldots, t_n\sigma))$ holds for every $(p, q) \in S$,
2. $r(A_0 \leftarrow A_1, \ldots, A_m, B_0 \leftarrow B_1, \ldots, B_m)$ holds if $r(A_i, B_i)$ holds for every $i, 0 \leq i \leq m$.

Now we can define $A$ as:

$A(x) = \{ y \mid r(x, y) \text{ holds} \}$.

The reflective semantics of this kind of analogical reasoning can be defined as follows. Given a logic program $(P, E)$, it can be divided into two subprograms, $(P_1, E)$ and $(P_1, E)$, as mentioned above. Let $U_{P_1}, B_{P_1}$ and $\text{pred}(P_1)$ (resp., $U_{P_1}, B_{P_1}, \text{pred}(P_1)$) be the Herbrand universe, the $E$-base and the set of predicate symbols of $P_1$ (resp., $P_1$). The mapping $T^{A}_{(P,E)}$, which allows the derivation of analogical consequences as outlined above, characterizes the consequences of $P_1$ with respect to the clauses of $P_1$ itself and the clauses of $P_2$.

7 Related work and concluding remarks

In Section 1 we gave general references to the ample subject of metalevel architectures and reflection and in Section 2 we reviewed the basic literature on this matter. In this section, we make an attempt to more specifically relate our approach to other proposals advanced in several contexts, since we wish to emphasize that it might be helpful, at least conceptually, to fulfill the needs arising in diverse problem domains such as software engineering, automated reasoning and theorem proving, knowledge representation and machine learning. Though the novelty of the proposed paradigm does not allow a direct comparison with other work, we will try to highlight possible commonalities with approaches having similar objectives put forward in different fields.

Several authors, especially in the logic programming community, have considered the utility of building program schemata that may represent a whole class of specific programs having a similar structure.

Kwok and Sergot [46] suggest ‘to write a logic program implicitly by stating the defining property which characterises it’ and show that ‘implicitly-defined programs may be used to simulate higher-order functions, define programs containing an infinite number of clauses and reuse existing programs’. They, however, ‘do not give specific proposals on how to extend existing languages by utilising this technique’.

Barker-Plummer [6] proposes an extension to the Prolog language to write commonly occurring program forms (called cliches) just once but to reuse them in a variety of ways, and implements this method by means of Prolog metaprogams.
Fuchs and Fromher [33] observe that ‘since the beginning of logic programming it has
been recognized that many logic programs ... are structured similarly, and can be understood
as instances of program schemata’. The objective is to transform an instance of one program
schema into an instance of another, to get a transformed program that is more efficient than
the original. The paper deals with transformation schemata which represent specific transforma-

tion strategies. Transformations generate equivalent programs in that the least Herbrand
model and the computed answers are preserved.

Yokomori [74] proposes logic program forms as sets of Horn clauses whose atoms may
have uninstantiated predicate name variables. An instantiation (called interpretation) of a
logic program form \( F \) is obtained by mapping the predicate name variables appearing in \( F \) to
predicate names, and from the variables appearing in \( F \) to terms, under suitable restrictions.
Instead of \( n \) programs having the same structure, one logic form can thus be given, together
with \( n \) interpretations. This is therefore a rather static approach, where neither a proof theory
nor a model theory is involved.

All of the above-mentioned approaches can be represented in Reflective Prolog, which in
turn is a particular instantiation of RCL as shown in Section 6.1.

Pfenning [58] calls `logical frameworks` a metalanguage for the specification of deduc-
tive systems, and argues that: `Logical frameworks are subject to the same general design
principles as other programming or specification languages. They should be as simple and
uniform as possible, yet they should provide concise means to express the concepts and meth-
ods of the intended application domain`. While surveying several frameworks, he remarks
that `research in logical framework is still in its infancy`.

We refer the reader to the literature mentioned in the introduction for many other meta-
level architectures, systems and languages that have been proposed, in particular those not
involving reflection that therefore we have not explicitly mentioned. The approach discussed
in the present paper differs from all of this work in that it is intended to show that, instead
of defining different architectures and languages for different knowledge representation, rea-
soning and learning tasks, it suffices to represent the latter as reflection principles in one and
the same single language, as we have attempted to show in the examples of Section 6.

Considering in particular the application of the general framework presented in this paper
to the field of metalogic languages, Reflective Prolog (Section 6.1) has been compared to
the other main approaches in [24]. A more recent approach, not considered there, is that of
[39], which is very similar to [24] about the treatment of naming and unification, except for
providing multiple theories, and names for theories. Theories are able to exchange formulae
that they can prove, by means of a distinguished binary predicate \( \text{demo} \), appearing explicitly
in the body of clauses, and having the name of a theory as the first argument, and the name of
a formula as the second argument. It is interesting to notice that this approach could be easily
modelled in \( RCL \): theory communication could be modelled as in Section 6.2, using \( \text{demo} \)
on both sides (instead of \( \text{tell/told} \)), and \( \text{demo} \) could be forced to convey provable formulae by
means of the reflection principle \( \text{Reflection up} \) of Section 6.1, with \( \text{demo} \) instead of \( \text{solve} \).

Finally, let us review how the present paper relates to our own previous work on the matter.

A language for building reflective, non-conservative extensions of Horn clause theories
was first proposed in [22] and fully defined formally in [24]. The system was then aug-
mented with a reflective, non-monotonic negation apt to represent non-monotonic reasoning
[23]. A formalization of analogical reasoning in this reflective logic was elaborated in [25].
Reflection was used to represent communication among different theories/agents in [10, 20].
The very idea that a common view underlying such diverse contexts could be systematized in
the unifying framework of reflection principles was first advanced in [21].

In order to achieve a more language-independent formulation of reflection principles, the system’s syntactical apparatus (language and proof theory) was then parametrized, using equational name theories for encoding facilities, and associated rewriting systems for substitution facilities [8, 9]. The present paper represents a new attempt to both clarify the role of reflection principles at the knowledge level and to formalize it at this enhanced technical level.

To summarize, the RCL system proposed in this paper is intended to be a logical framework:

- theoretically well founded with proved semantic properties;
- carefully designed in both the basic features and the flexible parametrical ones;
- fully worked out in all the technical details;
- practically implementable with known state-of-the-art techniques;
- wide in scope with respect to the set of tasks representable with it (from software engineering to knowledge representation, from common-sense reasoning to theorem proving);
- based on a single concept (the proposed form of reflection principles) for uniformly addressing these different tasks and domains;
- aimed at two classes of potential users: (i) those who may find its basic default features sufficient for their applications and by sticking to them are guaranteed with respect to soundness and completeness and (ii) those who may wish to exploit its constructive parametrical features to experiment with tailored forms of encodings and resolution reflection principles for more sophisticated applications and accept the burden of checking the holding of the required semantic properties.

We now wish to conclude the paper with a disclaimer. We believe reflection to be a powerful concept, yet a difficult one both theoretically and for practical implementations. Our system is limited to the extent that it is based on enhanced Horn clauses (not full first-order logic) for both language and metalanguage, with the same inference rule (SLD-resolution), which is different from other approaches that use distinct languages and/or inference systems. We are aware that the system we have proposed is just one single point in a huge space of possibilities, largely still to be explored. Some steps have been taken very recently towards establishing a groundwork for comparing different kinds of reflection and for studying their underlying theoretical properties [18, 54]. Our contribution is an effort to include reflection in the reconciliation of logic and computation that we feel is very much to be in the spirit and (we may say by now) the tradition of computational logic and logic programming.

In the next future, RCL will be fully implemented, taking as a starting point the existing implementation of Reflective Prolog, which is fully working, and has been used in several applications.

References
Reflection Principles in Computational Logic


Reflected Principles in Computational Logic


Appendix

Proposition 4.5 Let \([P, E]\) be a logic program and \(\ni A_1, \ldots, A_k\) a ground definite goal. Then \([P, E] \cup \{\ni A_1, \ldots, A_k\}\) is reflectively \(E\)-unsatisfiable if and only if \(A_1 \land \ldots \land A_k\) is a reflective logical \(E\)-consequence of \([P, E]\).

Proof. Suppose that \([P, E] \cup \{\ni A_1, \ldots, A_k\}\) is reflectively \(E\)-unsatisfiable. Let \(I\) be any \(E\)-interpretation of \([P, E]\). Assume that \(I\) is a reflective \(E\)-model of \([P, E]\). As \([P, E] \cup \{\ni A_1, \ldots, A_k\}\) is reflectively \(E\)-unsatisfiable, \(I\) cannot be a reflective \(E\)-model of \(\neg\{A_1 \land \ldots \land A_k\}\). Hence, each atom \(A_i, 1 \leq i \leq k\), is true under \(I\), i.e. \(I\) is a reflective \(E\)-model for every \(A_i\). Consequently \(A_1 \land \ldots \land A_k\) is a reflective logical \(E\)-consequence of \([P, E]\).

Conversely, suppose that \(A_1 \land \ldots \land A_k\) is a reflective logical \(E\)-consequence of \([P, E]\). Let \(I\) be an \(E\)-interpretation of \([P, E]\) and assume that \(I\) is a reflective \(E\)-model of \([P, E]\). Then \(I\) is also a reflective \(E\)-model of \(A_1 \land \ldots \land A_k\). Hence, \(I\) is not a reflective \(E\)-model of \(\neg\{A_1 \land \ldots \land A_k\}\). Consequently, \([P, E] \cup \{\ni A_1, \ldots, A_k\}\) is reflectively \(E\)-unsatisfiable. \(\blacksquare\)
Proposition 4.7 Let $\mathcal{R}$ be a reflection principle and $I$ an $E$-interpretation of a logic program $\langle P, E \rangle$. The mapping $T^{\mathcal{R}}_{(P, E)}$ is continuous.

PROOF. Let $X$ be a subset of $2^{\mathcal{R}(P, E)}$. Notice first that if $\{A_1, \ldots, A_m\} \subseteq \text{lab}(X)$ implies $\{A_1, \ldots, A_m\} \subseteq I$, for some $I \in X$. In order to show that $T^{\mathcal{R}}_{(P, E)}$ is continuous, we have to show that $T^{\mathcal{R}}_{(P, E)}(\text{lab}(X)) = \text{lab}(T^{\mathcal{R}}_{(P, E)}(X))$, for each directed subset $X$ of $2^{\mathcal{R}(P, E)}$. Now we have that

$$[A] \in T^{\mathcal{R}}_{(P, E)}[\text{lab}(X)]$$

iff $\{A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m\} \in \text{ground}(P \cup \mathcal{R}(P))$, $E \models e_i$ for all $i, 1 \leq i \leq q$, and $\{A_1, \ldots, A_m\} \subseteq \text{lab}(X)$.

iff $\{A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m\} \in \text{ground}(P \cup \mathcal{R}(P))$, $E \models e_i$ for all $i, 1 \leq i \leq q$, and $\{A_1, \ldots, A_m\} \subseteq I$, for some $I \in X$.

iff $[A] \in T^{\mathcal{R}}_{(P, E)}(I)$, for some $I \in X$.

iff $[A] \in \text{lab}(T^{\mathcal{R}}_{(P, E)}(X))$.

\hfill \Box

Proposition 4.8 Let $\mathcal{R}$ be a reflection principle and $I$ an $E$-interpretation of a logic program $\langle P, E \rangle$. Then $I$ is a reflective $E$-model of $\langle P, E \rangle$ if and only if $T^{\mathcal{R}}_{(P, E)}(I) \subseteq I$.

PROOF. $I$ is a reflective $E$-model for $\langle P, E \rangle$ iff the following two cases hold.

Case 1

For every $\{A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m\} \in \text{ground}(P)$, we have that $E \models e_i$ for all $i, 1 \leq i \leq q$, and $\{A_1, \ldots, A_m\} \subseteq I$ implies that $[A] \in I$ because $I$ is an $E$-model of each clause in $P$;

iff $T^{\mathcal{R}}_{(P, E)}(I) \subseteq I$.

Case 2

For every $\{A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m\} \in \text{ground}(\mathcal{R}(P))$, we have that $E \models e_i$ for all $i$, $1 \leq i \leq q$, and $\{A_1, \ldots, A_m\} \subseteq I$ implies that $[A] \in I$ because, by the definition of reflective $E$-model, $I$ is an $E$-model of each reflective axiom in $\mathcal{R}(P)$;

iff $T^{\mathcal{R}}_{(P, E)}(I) \subseteq I$.

\hfill \Box

Theorem 4.12 Let $\langle P, E \rangle$ be a logic program and $\leftarrow A_1, \ldots, A_k$ a definite goal. Suppose that $\langle H, F \rangle$ is an answer for $\langle P, E \rangle \cup \leftarrow A_1, \ldots, A_k$ and $H'$ is an $E$-solution of $F$. If $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is ground, then the following are equivalent:

(a) $\langle H, F \rangle$ is a correct answer.

(b) $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is true w.r.t. every reflective $E$-model of $\langle P, E \rangle$.

(c) $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is true w.r.t. the least reflective $E$-model of $\langle P, E \rangle$.

PROOF. (a) $\Rightarrow$ (c)

By the definition of correct answer,

(c) $\Rightarrow$ (b) $\Rightarrow$ (a)

$\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is true w.r.t. the least reflective $E$-model of $\langle P, E \rangle$

implies $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is true w.r.t. all reflective $E$-models of $\langle P, E \rangle$

implies $\neg\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is false w.r.t. all reflective $E$-models of $\langle P, E \rangle$

implies $\langle P, E \rangle \cup \{\neg\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'\}$ has no reflective $E$-models

implies $\langle P, E \rangle \cup \{\neg\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'\}$ is reflectively $E$-unsatisfiable

implies $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is a reflective logical $E$-consequence of $\langle P, E \rangle$ by Proposition 4.5 since $\langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{H}'$ is ground

implies $\langle H, F \rangle$ is correct because $H'$ is an $E$-solution of $F$.

\hfill \Box
Theorem 5.2 (Soundness of SLD\textsuperscript{R}-resolution) Let \( \{P, E\} \) be a logic program and \( G \) a definite goal. Every computed answer for \( \{P, E\} \cup \{G\} \) is a correct answer for \( \{P, E\} \cup \{G\} \).

Proof. Let \( G \) be a goal of the form \( \leftarrow A_1, \ldots, A_k \), and let \( \langle H_n, F_n \rangle \) be the pair containing the Herbrand assignment and the set of name equations of the \( n \)-th step of the SLD\textsuperscript{R}-refutation of \( \{P, E\} \cup \{G\} \). In order to prove our theorem, we have to show that, for every \( E \)-solution \( H' \) of \( F_n \), \( \forall (G_{H_n} H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \). The result is proved by induction on the length \( n \) of the SLD\textsuperscript{R}-refutation.

Base case \((n = 1)\)

This means that \( G \) is a goal of the form \( \leftarrow A_1 \). The initial state is \( \{\{A_1\}, \{\}\} \). We distinguish between two cases.

Case 1

\( A_1 \) is an atom of the form \( p(t_1, \ldots, t_k) \). \( P \) has a unit clause of the form \( \langle \{\}, \{t_1 = t'_1, \ldots, t_k = t'_k\} \rangle \implies \langle H_1, \{\} \rangle \). Note that, as unit clauses do not contain occurrences of \( \uparrow \) and \( \downarrow \), the set \( F_1 \) of name equations is the empty set. As \( \implies \) is sound for \( E \), \( p(t_1, \ldots, t_k) H_1 \) is an instance of \( p(t'_1, \ldots, t'_k) \). Thus, \( \forall (p(t_1, \ldots, t_k) H_1) \) is a logical \( E \)-consequence of \( \{P, E\} \) and, therefore, also a reflective logical \( E \)-consequence of \( \{P, E\} \).

Inductive step

Suppose that the result holds for computed answers coming from SLD\textsuperscript{R}-refutations of length \( n - 1 \), and consider a refutation of length \( n \). Let \( A_m \) be the selected atom in \( G \) and \( H' \) be an \( E \)-solution of \( F_n \). We distinguish between two cases.

Case 1

\( A_m \) is \( p(t_1, \ldots, t_k) \), \( p(t'_1, \ldots, t'_k) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r \) \((q \geq 0, r \geq 0)\) is a clause in \( P \) and \( \langle H_{n-1}, F_{n-1}; \{t_1 = t'_1, \ldots, t_k = t'_k, e_1, \ldots, e_q\} \rangle \implies \langle H_n, F_n, \{\} \rangle \). By soundness of \( \implies \) and by the induction hypothesis, \( \forall (\langle A_1 \land \ldots \land A_{m-1} \land B_1 \land \ldots \land B_r \land A_{m+1} \land \ldots \land A_k \rangle H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \). We prove our claim by considering two distinct subcases depending on whether \( r = 0 \) or \( r > 0 \).

Subcase \((r = 0)\)

Since by soundness of \( \implies \) \( E \models \forall (\langle e_1 \land \ldots \land e_q \rangle H_n H') \), \( \forall (p(t'_1, \ldots, t'_k) H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \). Thus, also \( \forall (p(t_1, \ldots, t_k) H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \), and consequently \( \forall (\langle A_1 \land \ldots \land A_k \rangle H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \).

Subcase \((r > 0)\)

Since by soundness of \( \implies \) \( E \models \forall (\langle e_1 \land \ldots \land e_q \rangle H_n H') \), \( \forall (\langle B_1 \land \ldots \land B_r \rangle H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \), \( \forall (p(t_1, \ldots, t_k) H_n H') \) is a reflective logical \( E \)-consequence of \( \{P, E\} \). Hence, \( \forall (\langle A_1 \land \ldots \land A_k \rangle H_n H') \) is also a reflective logical \( E \)-consequence of \( \{P, E\} \).

Case 2

\( A_m \) is of the form \( p(t_1, \ldots, t_k) \). \( C \) is a clause in \( P \), \( \mathcal{R}(C) \) contains a clause of the form \( \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r \) \((q \geq 0, r \geq 0)\), and \( \langle H_{n-1}, F_{n-1}; \{t_1 = t'_1, \ldots, t_k = t'_k, e_1, \ldots, e_q\} \rangle \implies \langle H_n, F_n, \{\} \rangle \). The proof of this case is similar to the proof of the previous case. ■
Reflection Principles in Computational Logic 783

Corollary 5.3 Let \( \{ P, E \} \) be a logic program and \( G \) a definite goal. Suppose that there exists an \( SLD^R \)-refutation of \( \{ P, E \} \cup \{ G \} \). Then \( \{ P, E \} \cup \{ G \} \) is reflectively \( E \)-unsatisfiable.

Proof. Let \( G \) be a goal of the form \( \leftarrow A_1, \ldots, A_k \). As \( \models \) is sound for \( E \), by Theorem 5.2, every computed answer \( \{ H, F \} \) of \( \{ P, E \} \cup \{ G \} \) is correct. Thus, for every \( E \)-solution \( H' \) of \( F \), \( \forall(\{ A_1 \land \ldots \land A_k \} \hat{H} \hat{H}') \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \). It follows that \( \{ P, E \} \cup \{ G \} \) is reflectively \( E \)-unsatisfiable.

Corollary 5.5 If a ground atom \( A \) belongs to the success set of a logic program \( \{ P, E \} \), then \( \{ A \} \) is contained in the least reflectively \( E \)-model of \( \{ P, E \} \).

Proof. Suppose that \( \{ P, E \} \cup \{ \leftarrow A \} \) has an \( SLD^R \)-refutation with computed answer \( \{ H, F \} \). As \( \models \) is sound for \( E \) and \( A \) is ground, by Theorem 5.2, \( A \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \). Hence \( \{ A \} \) is in the least reflectively \( E \)-model of \( \{ P, E \} \).

Theorem 5.7 Let \( \{ P, E \} \) be a logic program and \( G \) a definite goal \( \leftarrow A_1, \ldots, A_k \). Suppose \( \{ P, E \} \cup \{ G \} \) has an \( SLD^R \)-refutation of length \( n \) with computed answer \( \{ H_n, F_n \} \). Then, \( \bigcup_{j=1}^{k} \Psi(\{ A \} \hat{H}_n \hat{H}') \subseteq T_{\{ P, E \}}^{\hat{\uparrow} \uparrow n} \) for every \( E \)-solution \( H' \) of \( F_n \).
Let $C$ be the selected clause in $P$. We have two subcases. (In the remaining part of the proof let $H'$ be an $E$-solution of $F_n$.)

Subcase (i)

$A_j$ is $p(t_1, \ldots, t_k)$. $C$ is $p(t'_1, \ldots, t'_k) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r$ ($q \geq 0, r \geq 0$), and

$H_{n-1}$, $F_{n-1}$, $\{t_1 = t'_1, \ldots, t_k = t'_k, e_1, \ldots, e_q\} \Rightarrow (H_n, F_n, \{\})$. By the soundness of $\Rightarrow$, $p(t_1, \ldots, t_k)\widehat{\cdot}H_n H'$ is an instance of $p(t'_1, \ldots, t'_k)$.

If $r = 0$, we have $\Psi(p(t'_1, \ldots, t'_k)\widehat{\cdot}H_n H') \subseteq T_P^{(P, E)}$. Thus $\Psi(p(t_1, \ldots, t_k)\widehat{\cdot}H_n H') = \Psi(p(t'_1, \ldots, t'_k)\widehat{\cdot}H_n H') \subseteq T_P^{(P, E)} \subseteq T_P^{(P, E)} \uparrow \uparrow 1$. If $r > 0$, by the induction hypothesis, $\Psi(B_r H_n H) \subseteq T_P^{(P, E)} \uparrow (n-1)$ for all $i$, $1 \leq i \leq r$. By the definition of $T_P^{(P, E)}$, we have that $\Psi(p(t_1, \ldots, t_k)\widehat{\cdot}H_n H') \subseteq T_P^{(P, E)} \uparrow \uparrow n$.

Subcase (ii)

$A_j$ is $p(t_1, \ldots, t_k)$. $p(t'_1, \ldots, t'_k) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r$ ($q \geq 0, r \geq 0$) is a clause in $R(C)$, and $H_{n-1}, F_{n-1}, \{t_1 = t'_1, \ldots, t_k = t'_k, e_1, \ldots, e_q\} \Rightarrow (H_n, F_n, \{\})$. The proof of this subcase is similar to the proof of subcase (i).

**Lemma 5.8** (Lifting lemma) Let $(P, E)$ be a logic program, $H$ a Herbrand assignment and $G$ a definite goal. Suppose there exists an SLDR-refutation of $(P, E) \cup \{G \hat{H}\}$ with success state $(\{\}, H_n, F_n)$. If $R_E$ is convergent, then there exists an SLDR-refutation of $(P, E) \cup \{G\}$ of the same length with success state $(\{\}, H'_n, F'_n)$ such that $(H'_n, F'_n, H) \Rightarrow (H_n, F_n, \{\})$.

**Proof.** Since $R_E$ is convergent, by Proposition 3.10, $\Rightarrow$ converges.

Note first that by the definition of convergent system, if

$$\langle H, F, A \cup B \rangle \Rightarrow^n_r \langle H_1, F_1, B \rangle \Rightarrow^n_r \langle H_2, F_2, \{\} \rangle$$

then

$$\langle H, F, A \cup B \rangle \Rightarrow^n_r \langle H'_1, F'_1, A \rangle \Rightarrow^n_r \langle H'_2, F'_2, \{\} \rangle.$$ Equivalently,

$$\langle H, F, A \rangle \Rightarrow (H_1, F_1, \{\}) \text{ and } (H_1, F_1, B) \Rightarrow (H_2, F_2, \{\}).$$

Now consider an SLD-refutation of $(P, E) \cup \{G \hat{H}\}$ instead of applying the substitution $\hat{H}$ to $G$, we equivalently, consider as initial state the state $(G, H, \{\})$. In the following, let $C_i$ and $S_i$ be the selected clause in $P$ and the set of equations needed to perform the $i$-th refutation step as defined in Definition 4.13. Thus, at the $i + 1$ step of the SLD-refutation of $(P, E) \cup \{G \hat{H}\}$ the state $(G, H_i, F_{i+1}, S_{i+1})$ is obtained from $(G, H_i, F_i)$ and $C_i$ if

$$\langle H_i, F_i, S_i \rangle \Rightarrow (H_i, F_i, \{\}) \text{ and } (H_i, F_i, B) \Rightarrow (H_{i+1}, F_i, \{\}).$$

Hence we have that:

$$\langle H, \{\}, S_1 \rangle \Rightarrow (H_1, F_1, \{\})$$

$$\langle H_1, F_1, S_2 \rangle \Rightarrow (H_2, F_2, \{\})$$

$$\cdots$$

$$\langle H_{n-1}, F_{n-1}, S_n \rangle \Rightarrow (H_n, F_n, \{\}).$$

As $H$ is a Herbrand assignment, i.e. a set of equations in solved form, the state

$$(\{\}, \{\}, H) \Rightarrow (H, \{\}, \{\}).$$
Finally, by convergency of $R_E$

$$\langle \{\}, \{\}, S_1 \cup \ldots \cup S_n \cup H \rangle \implies \langle H_n, F_n,\{\}\rangle$$

or, equivalently,

$$\langle \{\}, \{\}, S_1 \cup \ldots \cup S_n \rangle \implies \langle H_n', F'_n, \{\}\rangle \text{ and } \langle H_n', F'_n, H \rangle \implies \langle H_n, F_n, \{\}\rangle.$$

Theorem 5.9 Let $\{P, E\}$ be a logic program. A ground atom $A$ belongs to the success set of $\{P, E\}$ if and only if $[A]$ is contained in the least reflective $E$-model of $\{P, E\}$.

Proof. By Corollary 5.5, it suffices to show that, if $[A]$ belongs to the least reflective $E$-model of $\{P, E\}$, then $A$ is contained in the success set of $\{P, E\}$. Suppose that $[A]$ is in the least reflective $E$-model of $\{P, E\}$. Then by Theorem 4.9, $[A] \in T_{\{P, E\}}^R \uparrow n$, for some $n \in \omega$. We prove by induction on $n$ that if $[A] \in T_{\{P, E\}}^R \uparrow n$, then $\{P, E\} \cup \{\leftarrow A\}$ has a SLD$_R$-refutation and hence $A$ is in the success set of $\{P, E\}$.

Base case ($n = 1$) 
This means that $[A] \in T_{\{P, E\}}^R \uparrow 1$. We distinguish between two cases.

Case 1
$A$ is a ground atom of the form $p(t_1, \ldots, t_k)$ and there exists a unit clause in $P$, say $p(t'_1, \ldots, t'_k) \leftarrow$, such that $E \models \exists(t_1 = t'_1 \land \ldots \land t_k = t'_k)$. By soundness of $\implies$, $\langle \{\}, \{\}, t_1 = t'_1, \ldots, t_k = t'_k \rangle \implies \langle H, \{\}, \{\} \rangle$, for some Herbrand assignment $H$. Then, by the definition of SLD$_R$-resolution (case 1), $\{P, E\} \cup \{\leftarrow p(t_1, \ldots, t_k)\}$ has an SLD$_R$-refutation.

Case 2
$A$ is a ground atom of the form $p(t_1, \ldots, t_k)$ and there exists a clause $C$ in $P$ such that $R(C)$ contains a unit clause of the form $p(t'_1, \ldots, t'_k) \leftarrow$ and $E \models \exists(t_1 = t'_1 \land \ldots \land t_k = t'_k)$. By soundness of $\implies$, $\langle \{\}, \{\}, t_1 = t'_1, \ldots, t_k = t'_k \rangle \implies \langle H, \{\}, \{\} \rangle$, for some Herbrand assignment $H$. Then, by the definition of SLD$_R$-resolution (case 2), $\{P, E\} \cup \{\leftarrow p(t_1, \ldots, t_k)\}$ has an SLD$_R$-refutation.

Inductive step
Suppose that the result holds for $n - 1$. Assume that $[A] \in T_{\{P, E\}}^R \uparrow n$, then by the definition of $T_{\{P, E\}}^R \uparrow n$, one of the following cases holds.

Case 1
$A$ is a ground atom of the form $p(t_1, \ldots, t_k)$ and there exists a ground instance $\langle p(t'_1, \ldots, t'_k) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_m \rangle \hat{H}$, $(q \geq 0, m \geq 0)$ of a clause in $P$ such that $E \models \exists(t_1 = t'_1 \land \ldots \land t_k = t'_k)$ and $\langle [B_1, \ldots, [B_m \hat{H}] \rangle \subset T_{\{P, E\}}^R \downarrow (n - 1)$, for some Herbrand assignment $H$.

Case 2
$A$ is a ground atom of the form $p(t_1, \ldots, t_k)$, $C$ is a clause in $P$ and there exists a ground instance $\langle p(t'_1, \ldots, t'_k) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_m \rangle \hat{H}$, $(q \geq 0, m \geq 0)$ of a clause in $R(C)$ such that $E \models \exists(t_1 = t'_1 \land \ldots \land t_k = t'_k) \land \exists(e_1, \ldots, e_q) \hat{H}$ and $\langle [B_1, \ldots, [B_m \hat{H}] \rangle \subset T_{\{P, E\}}^R \downarrow (n - 1)$, for some Herbrand assignment $H$.

Thus, by the induction hypothesis, $\{P, E\} \cup \{\leftarrow A \hat{H}\}$, $1 \leq i \leq m$, has an SLD$_R$-refutation. Because each $B_i \hat{H}$ is ground, these refutations can be combined into a refutation of $\{P, E\} \cup \{\leftarrow B_1, \ldots, B_m \hat{H}\}$. Hence $\{P, E\} \cup \{\leftarrow A \hat{H}\}$ has an SLD$_R$-refutation and we can apply the Lifting lemma to obtain an SLD$_R$-refutation of $\{P, E\} \cup \{\leftarrow A\}$.

Theorem 5.10 Let $\{P, E\}$ be a logic program and $G$ a definite goal. Suppose that $\{P, E\} \cup \{G\}$ is reflectively $E$-unsatisfiable. Then there exists an SLD$_R$-refutation of $\{P, E\} \cup \{G\}$.
PROOF. Let \( G \) be the goal \( \leftarrow A_1, \ldots, A_k \). As \( \{ P, E \} \cup \{ G \} \) is reflectively \( E \)-unsatisfiable, \( G \) is false w.r.t. the least reflective \( E \)-model \( M^E_{\{ P, E \}} \). Hence there exists some ground instance \( \hat{G} \) of \( G \) such that \( \neg \left( \left[ A_1 \hat{H} \right] \land \ldots \land \left[ A_k \hat{H} \right] \right) \) is false w.r.t. \( M^E_{\{ P, E \}} \). Thus \( \left\{ \left[ A_1 \hat{H} \right], \ldots, \left[ A_k \hat{H} \right] \right\} \subseteq M^E_{\{ P, E \}} \). By Theorem 5.9, there is an SLD\( R \)-refutation for \( \{ P, E \} \cup \{ \leftarrow A_i \hat{H} \} \) for each \( i, 1 \leq i \leq k \). As each \( A_i \hat{H} \) is ground, its computed answer \( \langle H_i, F_i \rangle \) contains variables that are distinct from the variables of the remaining computed answers. Thus we can combine these refutations into a refutation for \( \{ P, E \} \cup \{ \langle \leftarrow A \hat{H} \rangle \} \). Finally, we apply the Lifting lemma.

Lemma 5.11 Let \( \{ P, E \} \) be a logic program and \( A \) an atom. Suppose that \( x_1, \ldots, x_n \) are all the variables and the metavariables occurring in \( A \) and that \( \forall x_1, \ldots, x_n A \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \). Then, there exists an SLD\( R \)-refutation of \( \{ P, E \} \cup \{ \leftarrow A \} \) with computed answer \( \langle H, F \rangle \) such that \( E \models \forall x_1, \ldots, x_n \exists \langle H \cup F \rangle \).

Proof. Let \( a_1, \ldots, a_n \) be distinct constants or metaconstants not appearing in \( \{ P, E \} \) or \( A \), and let \( \hat{H} \) be the Herbrand assignment \( \{ x_1 = a_1, \ldots, x_n = a_n \} \) (we assume that for all \( i, 1 \leq i \leq n \), if \( a_i \) is a constant, then \( x_i \) is a variable, and vice versa, if \( a_i \) is a metaconstant, then \( x_i \) is a metavariable). Then \( A \hat{H} \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \). As \( A \hat{H} \) is ground, Theorem 5.9 states that \( \{ P, E \} \cup \{ \leftarrow A \hat{H} \} \) has an SLD\( R \)-refutation.

As the \( a_i \) do not appear in \( \{ P, E \} \) or \( A \), by replacing \( a_i \) by \( x_i \) for all \( i, 1 \leq i \leq n \), in this refutation, we obtain an SLD\( R \)-refutation of \( \{ P, E \} \cup \{ \leftarrow A \} \) with computed answer \( \langle H, F \rangle \) such that the bindings in \( H \) for \( x_1, \ldots, x_n \) are variable-pure.

Furthermore, the equations in \( F \) are always satisfied independently from the values of \( x_1, \ldots, x_n \), as they are satisfied in the SLD\( R \)-refutation for \( \{ P, E \} \cup \{ \leftarrow A \hat{H} \} \) when substituted by arbitrary constants, i.e. \( a_1, \ldots, a_n \). Hence, it holds that \( E \models \forall x_1, \ldots, x_n \exists \langle H \cup F \rangle \).

Theorem 5.13 (Completeness of SLD\( R \)-resolution) Let \( \{ P, E \} \) be a logic program and \( G \) a definite goal. If \( R \) is convergent, then for every correct answer \( \langle H, F \rangle \) for \( \{ P, E \} \cup \{ G \} \), there exists a computed answer \( \langle H', F' \rangle \) for \( \{ P, E \} \cup \{ G \} \). Furthermore, there exists a Herbrand assignment \( H'' \) such that, for every \( E \)-solution \( H_P \) and \( H_{P'} \) of \( F \) and \( F' \), respectively, \( \langle G \hat{H} \rangle \hat{H}\hat{P} = G \hat{H} \hat{P} \) holds.

Proof. Suppose that \( G \) is the goal \( \leftarrow A_1, \ldots, A_k \). As \( \{ H, F \} \) is a correct answer, \( \forall \langle A_1 \land \ldots \land A_k \rangle \hat{H} \hat{P} \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \) for every \( E \)-solution \( H_P \) of \( F \). By Lemma 5.11, there exists an SLD\( R \)-refutation of \( \{ P, E \} \cup \{ \leftarrow A_i \hat{H} \hat{P} \} \) with computed answer \( \langle H_i, F_i \rangle \) for all \( i, 1 \leq i \leq k \). Let \( H_P \) be an \( E \)-solution of \( F_i \), \( 1 \leq i \leq k \). As \( \hat{H}_i \hat{P}_{H_i} \) does not instantiate any of the variables in \( A_i \) and the variables in every SLD\( R \)-refutation are standardized apart, we can combine these SLD\( R \)-refutations into an SLD\( R \)-refutation of \( \{ P, E \} \cup \{ G \hat{H} \hat{P} \} \).

Assume the computed answer for \( \{ P, E \} \cup \{ G \hat{H} \hat{P} \} \) is \( \langle H''', F''' \rangle \). As \( \forall \langle G \hat{H} \hat{P} \rangle \) is a reflective logical \( E \)-consequence of \( \{ P, E \} \), by Lemma 5.11, \( H''', \hat{P}_{H'''} \) does not instantiate any variable in \( \hat{H} \hat{P} \). Thus, \( G \hat{H} \hat{P} = G \hat{H}'''' \hat{P}_{H'''} \) holds.

By the Lifting lemma, there exists an SLD\( R \)-refutation of \( \{ P, E \} \cup \{ G \} \) with success state \( \langle \{ \}, H', F' \rangle \) such that \( \langle H', F', H \cup H_P \rangle \models H''', F''' \). Let \( H''' \) be \( H \cup H_P \). Then \( G \hat{H}'''' \hat{P}_{H'''} = G \hat{H}'''' \hat{P} = G \hat{H}'''' \hat{P} \) holds.

Received 19 December 1997