Reflection Principles in Computational Logic

Pierangelo Dell'Acqua

A Dissertation submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Uppsala University.

Uppsala University
Computing Science Department
Box 311, S-751 05 Uppsala, Sweden

Uppsala Theses in Computing Science 30
ISSN 0283-359X
ISBN 91-506-1298-0

(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 1998)

Abstract

Dell'Acqua, P. 1998: Reflection Principles in Computational Logic, *Uppsala Theses in Computing Science 30.* 154pp. Uppsala. ISSN 0283-359X, ISBN 91-506-1298-0.

We introduce the concept of reflection principles 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.

The proposed approach to reflection is powerful and flexible enough to be integrated into different frameworks. We show how the use of reflection principles can be integrated into a framework of rational, reactive agents to enhance their reasoning capabilities. Finally, relationship with a variety of distinct sources within the literature on relevant topics is discussed.

Pierangelo Dell'Acqua, Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden.

ISSN 0283-359X
ISBN 91-506-1298-0
Printed by Kopieringshuset, Uppsala 1998.

To Aida and Samantha

Acknowledgments

Many people have helped me during my work with this thesis. First and foremost I wish to express my sincere thanks to my advisors *Dr. Jonas Barklund*, *Dr. Stefania Costantini* and *Professor Gaetano Aurelio Lanzarone* for their support, advice and encouragement during these years.

Many thanks to my colleagues at the Computing Science Department in Uppsala for useful commenting on earlier versions of this thesis. In particular, I am indebted to *Dr. Anatoli Degtyarev*, *Dr. Anna Gomolińska*, *Dr. Sven-Olof Nyström*, *Dr. Margus Veanes* and *Eng. Aida Vitória* for numerous helpful and pertinent comments.

Moreover, I would like to express my gratitude to *Professor Alberto Pettorossi* and *Dr. Maurizio Proietti* for their advices and for posing crucial questions on naming relations during my visits at CNR in Rome.

The numerous discussions on agents with *Professor Robert Kowalski*, *Dr. Francesca Toni* and *Dr. Fariba Sadri* at the Imperial College have been most enlightening. I wish also to thank *Dr. Kung Kiu Lau* for useful discussions when he hosted me in Manchester, and *Professor Andrzej Skowron* and *Dr. Anna Gomolińska* for their hospitality during a visit at Warsaw.

This research was supported financially first by Università degli Studi di Milano and later by the Swedish National Board for Technical and Industrial Development (NUTEK) under contract No. 92-10452 (ESPRIT BRP 6810: *Computational Logic 2*) and the Swedish Research Council for the Engineering Sciences (TFR) under contract No. 95-304 (*Metaprogramming and metareasoning in logic*).

Contents

Chapter 1

Introduction

1.1 OVERVIEW AND OUTLINE OF THE THESIS

Reflective 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 [90, 91] and in theorem proving [56], in computer science [45, 85, 105], and in logic programming [11, 58, 77] has been generally recognised. (See also [1, 17, 24, 50, 92] for snapshots of research.)

The common intuitive notion of reflection in such different areas is that of an access relationship between theories or programs at the object level and theories 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. This is introduced and discussed in Chapter 2.

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' " [107]. 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.

Step I In *RCL*, the first step for specifying a deductive system (*DS*) is that of defining its naming device (encoding). Encodings are formalised 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. The breakthrough idea of the approach is the introduction of reflection principles for defining inference rules of *DS*. In particular, the user is required to express an inference rule R as a function $\mathcal{R}$, 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 $\mathcal{R}$, 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 $\mathcal{R}$. Consequently, the model-theoretic and fixed point semantics of T under $\mathcal{R}$ 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 order to do so, *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 clearly 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 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 Chapters 3 and 4.

For deduction, the concept of reflection principle is introduced in Section 5.1, and its integration into the declarative and procedural semantics of Horn clause theories is discussed in Sections 5.2 and 5.3.

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 5.2 and 5.3. Results of soundness and completeness of the proof theory with respect to the model theory are given in Chapter 6.

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 Chapters 7, 8 and 9, where three domains are exemplified and treated as case studies.

The first deductive system that we define (Chapter 7) is a metalogic programming language 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.

The second deductive system is able to represent agents and cooperation between multiple agents (Chapter 8). 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 (Chapter 9) 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.

The approach to agents based on reflection principles (Chapter 8) is powerful and flexible enough to be integrated into different frameworks. After having introduced the basic concepts and terminology (Chapter 10), we show (Chapter 11) how the reflection principles proposed to model introspective, communicative agents can be integrated into the framework of rational, reactive agents proposed by Kowalski and Sadri. The resulting kind of agents combine the characteristics of both.

The thesis is concluded in Chapter 12, 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.

1.2 SOURCE MATERIAL

This thesis is mainly based on the following material.

- Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Reflective Agents in Metalogic Programming, in: A. Pettorossi (ed.), *Meta-Programming in Logic*, LNCS 649, Springer-Verlag, Berlin, 1992.
- Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Extending Horn Clause Theories by Reflection Principles, in: C. MacNish, D. Pearce and L. M. Pereira (eds.), *Logics in Artificial Intelligence*, LNAI 838, Springer-Verlag, Berlin, 1994.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., SLD-Resolution with Reflection, in: M. Bruynooghe (ed.), *Logic Programming - Proc. 1994 Intl. Symp.*, MIT Press, Cambridge, Mass., 1994.
- Barklund, J., Boberg, K. and Dell'Acqua, P., A Basis for a Multilevel Metalogic Programming Language, in: L. Fribourg and F. Turini (eds.), *Logic Program Synthesis and Transformation - Meta-Programming in Logic*, LNCS 883, Springer-Verlag, Berlin, 1994.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Integrating Reflection into SLD-Resolution, in: A. Momigliano and M. Ornaghi (eds.), *Proc. Post-Conf. Ws. on Proof-Theoretical Extensions of Logic Programming*, 1994.

- Barklund, J., Boberg, K., Dell'Acqua, P. and Veanes, M., Meta-programming with Theory Systems, in: K. Apt and F. Turini (eds.), *Meta-logics and Logic Programming*, MIT Press, Cambridge, Mass., 1995.
- Dell'Acqua, P., SLD-Resolution with Reflection, PhL Thesis, Uppsala University, Uppsala, 1995.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Semantical Properties of Encodings in Logic Programming, in: J. Lloyd (ed.), *Logic Programming - Proc. 1995 Intl. Symp.*, MIT Press, Cambridge, Mass., 1995.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Multiple Metareasoning Agents for Flexible Query-Answering Systems, in: H. Christiansen, H. L. Larsen and T. Andreasen (eds.), *Proc. Ws. Flexible Query-Answering Systems*, 1996.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Metareasoning Agents for Query-Answering Systems, in: T. Andreasen, H. Christiansen and H. L. Larsen (eds.), *Flexible Query-Answering Systems*, Kluwer Academic Publishers, Boston, Mass., 1997.
- Dell'Acqua, P. Sadri, F. and Toni, F., Combining introspection and communication with rationality and reactivity in agents, To appear in: *Logic in Artificial Intelligence* (Jelia'98), LNAI, Springer-Verlag, 1998.
- Barklund, J., Costantini, S., Dell'Acqua, P. and Lanzarone, G. A., Reflection Principles in Computational Logic, Submitted to J. of Logic and Computation, 1997.

The order of the authors is purely alphabetical and is not intended to indicate the extent of the individual contributions. The main ideas behind some of the results that are presented in the thesis have been obtained in collaboration with the coauthors.

I thank my coauthors for generously allowing me to use parts of the material where no obvious borderlines exist between the contributions of different authors.

Chapter 2

Background

We introduce basic concepts and terminology.

2.1 METAPROGRAMMING IN LOGIC

Logic Programming is a style of programming based on formal logic. It is born from the idea that logic can be used not only for expressing problems, but also for solving them. Logic has traditionally been focussed upon the problem of determining whether a given conclusion (or theorem) C is logically implied by a set of assumptions (or theory) T. In the context of logic programming we are rather interested in computing solutions. Given a logic program (corresponding in logical terms to a theory) P and a goal statement (corresponding to a theorem) G, we are interested in determining a solution when proving G from P, where a solution σ is a set of bindings for the variables appearing in G such that $G\sigma$ is logically implied by P.

Logic programming is traditionally based on a subset of first order predicate calculus, i.e., Horn clause logic. This restriction, which leads to a loss of expressive power, has two advantages. It allows the use of efficient inference rules such as SLD-resolution. Furthermore, from a theoretical point of view, logical consequences can be determined by examining a single model, called the least Herbrand model. On the other hand, Horn clause logic has many limitations, and to overcome them a number of both *non-logical* and *meta-logical* features have been added to logic programming languages. The former features are called non-logical because they are dependent on the procedural behaviour of the program and, although important from a practical point of view, they violate the declarative semantics of logic. The latter features allow us to access and manipulate linguistic constructs, such as programs, goals and proofs, viewed as data structures. Meta-logical features are important in many types of programs. In fact, many applications of logic, such as applications that formalise proof procedures (metainterpreters) and knowledge assimilators, are metalogical. This has led to the study and development of powerful metaprogramming techniques that allow us to extend and modify the semantics of an existing object language.

Metaprogramming has so far been employed in many applications including debuggers, compilers and program transformers [53, 109, 112]. Furthermore, applications which can be formalised using other logics, such as modal logic of knowledge and belief [34, 69], and applications for theory construction [21] are objects of metaprogramming. A kind of architecture particularly suitable for metaprogramming is presented in the next section.

2.2 METALEVEL ARCHITECTURES

The advantages obtained from the ability to explicitly express and use metalevel knowledge have been widely recognised, especially in the AI literature. Aiello et al. [2], for example, argue that metaknowledge and metalevel reasoning is suitable for devising proof strategies in automated deduction systems, for controlling the inference in problem solving, and for increasing the expressive power of knowledge representation languages.

The need to formally represent knowledge and metaknowledge has led to the study of *metalevel architectures* [3], where these two kinds of knowledge are explicitly represented. The basic features of a metalevel architecture can be summarised in:

1. the *naming relation*, providing names at the metalevel for the expressions of the object level theory;
2. the metalevel formalisation of the object level properties;
3. the *linking rules*, that establish the connection between object level and metalevel in the inference process.

As the metalanguage is used to formalise properties of the object level theory, predicates in the metalanguage must take some representations of expressions of the object level theory as arguments. This is achieved by establishing a relationship, called the naming relation, between metalevel symbols and object level expressions.

At the metalevel, predicates can be used to define notions such as those of truth and provability. The notion of provability, for example, is typically formalised through a predicate $Demo(\ulcorner T \urcorner, \ulcorner \alpha \urcorner)$, that expresses at the metalevel the derivability of the object level formula α in the object level theory T, named $\ulcorner \alpha \urcorner$ and $\ulcorner T \urcorner$, respectively.

The distinguished metalevel predicates, which express properties of the object level, are related to the object level itself through linking rules. The most widely used linking rules take the form of inference rules:

$$\frac{T \vdash_O \alpha}{Pr \vdash_M Demo(\ulcorner T \urcorner, \ulcorner \alpha \urcorner)} \quad \text{and} \quad \frac{Pr \vdash_M Demo(\ulcorner T \urcorner, \ulcorner \alpha \urcorner)}{T \vdash_O \alpha}$$

where Pr is a metalevel description of provability at the object level. The first linking rule, *object-to-meta reflection*[1], allows one to assert the provability of $Demo(\ulcorner T \urcorner, \ulcorner \alpha \urcorner)$ in the metatheory when α is provable in the object theory T. The second linking rule, *meta-to-object reflection*, allows one to assert the provability of α in the object level theory T whenever it is possible to prove $Demo(\ulcorner T \urcorner, \ulcorner \alpha \urcorner)$ in the metatheory.

The application of reflection in the inference process can be *explicit* or *implicit*. In explicit reflection, the application of the linking rules is specified in the theory, i.e., it is necessary to specify in advance where to change the level during deduction. In implicit reflection, on the other hand, the linking rules are integrated in the procedural semantics of the metalevel architecture. Different architectures are then characterized by different conditions for applying reflection.

Metalevel architectures can be classified into two main categories: *amalgamated* and *separated*. We consider an architecture to be amalgamated when sentences of the object language and sentences of the metalevel are defined within a single theory. Therefore, mixed sentences combining object level and metalevel expressions are allowed. The main amalgamated approaches in the AI literature are established by Bowen & Kowalski [19], Attardi & Simi [9], Costantini & Lanzarone [38] and Lloyd et al. [60].

In separated architectures, the object theory and the metatheory are distinct theories. Therefore, it is not possible to write sentences mixing object level and metalevel expressions. The connection between the metatheory and the object theory is provided by the naming relation and the linking rules. Among the separated approaches we mention the FOL system by Weyhrauch [121] and 'Log by Cervesato & Rossi [27] (for a survey of these approaches see [4]).

On one hand, amalgamated approaches may turn out to be inconsistent. Tarski, for example, showed that if the object level and the metalevel are combined in the same language, then it is possible to express paradoxes such as the liar paradox. (See [117] for a brief discussion of the most important results achieved by logicians in this area.) On the other hand, separated approaches have less expressive power than amalgamated ones. Consider, for example, the sentence "A person is innocent if he or she cannot be proved guilty" (this example is taken from [19]). This can be formulated as:

$$Innocent(x) \leftarrow Person(x), not\, Demo(Facts, \ulcorner Guilty(x) \urcorner), Relevant(Facts)$$

Its formalisation combines a metalanguage condition with an object language condition and conclusion. Here *Facts* names the relevant facts and

[1]Some authors call these reflection principles upward and downward reflection, respectively; other authors, instead, use upward reflection (resp., downward reflection) to indicate the procedural shift from the object level to the metalevel (resp., from the metalevel to the object level). Therefore, we introduce new names to avoid such ambiguities.

assumptions which can be used in the attempt to establish guilt. Notice that a person might be actually guilty but not provable guilty.[2] In such a case the single level sentence

$$\mathit{Innocent}(x) \leftarrow \mathit{Person}(x), \mathit{not}\ \mathit{Guilty}(x)$$

would not lead to the conclusion that he or she is innocent, whereas the mixed object language and metalanguage sentence would.

Thus, separated architectures are limited to express reasoning with one level of nesting, while many AI applications (e.g., applications requiring the formalisation of epistemic notions such as knowledge and belief) often require deeper levels of nesting (e.g., $\mathit{Demo}(\mathit{Facts}, \ulcorner \mathit{Belief}(\mathit{John}, \ulcorner \mathit{Belief}(\ldots)\urcorner)\urcorner)$. See, for example, the issues raised by Moore [84]).

2.3 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, e.g., Suppes [111] for a discussion on this topic). Consider the following statements:

California is a state. (2.1)

California has ten letters. (2.2)

'California' is a state. (2.3)

'California' has ten letters. (2.4)

The statements (2.1) and (2.4) are true, while (2.2) and (2.3) 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 (2.3). 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 (2.3) can be clarified by rephrasing it as:

The word 'California' is a state.

(2.3) is about a word which (2.1) contains, and (2.1) is about no word at all, but a state. In (2.1) the state-name is *used*, while in (2.4) a quotation is used and the state-name is *mentioned*. To mention California we use

[2]Notice that some system may assert that a person is guilty without giving any evidence (*Facts*) of his guilt.

'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

Jeremiah = *'California'.* (2.5)

Then the following statements could be true,

Jeremiah is a name of a state. (2.6)

Jeremiah has ten letters. (2.7)

'Jeremiah' has eight letters. (2.8)

while the next is false

'Jeremiah' is a name of a state. (2.9)

Statement (2.9) could be rendered true by inserting another 'name of' in it

'Jeremiah' 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 (2.4) or phonetic and grammatical properties as in the following.

'Boston' is disyllabic. (2.10)

'Boston' is a noun. (2.11)

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

'Boston' designates the capital of Massachusetts. (2.12)

'Boston' is synonymous with 'the capital of Massachusetts'. (2.13)

Notice that in (2.13) quotations can be synonymous, while places cannot.

As Quine points out [95], 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 [26] for further discussion).

Quotations can also be applied to non-atomic expressions. For example, to state 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:

'Margus is Estonian' is true. (2.14)

but never

Margus is Estonian is true. (2.15)

(2.14) is a statement, while (2.15) is not. Notice that in (2.14) 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:

For every p, 'p' is the sixteenth letter of the alphabet. (2.16)

This sentence can be considered to be true, and the quantifier *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 *Margus is Estonian*, the false assertion:

'Margus is Estonian' is the sixteenth letter of the alphabet. (2.17)

Tarski [113], 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.

For example, the metalogical programming language *Reflective Prolog* [36, 38] implements a structural descriptive naming relation. The *Gödel* programming language [60] has names as an abstract data type, but provides enough predicates for testing and constructing names, and for extracting their constituents. Thus, the naming relation of *Gödel* can also be classified as structural descriptive.

2.4 NAMING RELATIONS

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 that one name. In practice, however, naming relations are typically functional and injective (see [115] 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 [38] for a discussion about possible uses of metaknowledge and metareasoning.

Although several AI systems with a metalevel architecture have implemented naming relations, the formal properties that they must satisfy have usually been assumed without motivations. Bowen & Kowalski [19], for instance, allow the naming relation to be non-functional and require it to be injective, while Hill & Lloyd [59] define a naming relation that is total, injective and functional.

So far, the only investigation of formal properties of naming relations is that of van Harmelen [116]. He argues that naming relations should be definable and meaningful. If the naming relation is a fixed part of a metalevel architecture, then it is not possible, in general, to find one that is optimal for all metalevel theories. In fact, the richness[3] of the naming relation determines not only the expressivity of the metatheory but also its computational complexity.[4] Therefore, the naming relation should be adapted to the particular

[3]The term richness is here used with the meaning of quantity of syntactic information encoded.

[4]In order to have an effective and efficient implementation of *'Log*, Cervesato & Rossi introduce in this language [27] two different but related metalevel representations (i.e., *quotation-mark* and *structural descriptive* names) for each syntactic object of the language. Thus, one can use quotation-mark names for efficiency, and structural descriptive names for obtaining more expressivity.

requirements of a given metatheory, and/or the application at hand. This motivates the choice, in our framework, to provide the encoding as a separately definable component. Furthermore, the naming relation should not only encode syntactic information, but it should also be useful for encoding pragmatic (use) and semantic (meaning) information, i.e., it should be meaningful (for some examples of meaningful naming relations see [116]).

This is in contrast with the approach discussed by Eshghi [46]. In fact, he argues that the purpose of the naming relation is to allow us to refer, in the metalanguage, to the relevant constructs of the object language. Thus, the naming relation must be

(i) *unambiguous*, i.e., if the names of two objects are the same, and they appear in the same context, those objects must be identical up to renaming of variables; and

(ii) *transparent*, i.e., the name of an object must be as similar to the object as possible in such a way to make (mixed object language and metalanguage) sentences more readable, and to aid implementation.

If the naming relation is *definable* instead of being a predefined component of a metalevel architecture, then it is necessary to state which formal properties it must satisfy, because these properties cannot be enforced by the system implementor and must instead be fulfilled at definition time.

Van Harmelen [116] only requires injectivity. He argues that the proper semantics, as pointed out by Tarski [113], is that the object theory should be regarded as a partial model of the metatheory, in the sense that the denotations of names at the metalevel should be the corresponding expressions at the object level, i.e., $[\![y]\!] = x$ whenever y is the name of x. This makes the naming relation the inverse relation of semantic denotation. Semantic denotation is required to be functional and the naming relation must therefore be injective.

Furthermore, van Harmelen argues that certain other properties might even be undesirable. Non-total naming relations allow information hiding between object level and metalevel. Because not all expressions can be described at the metalevel, they are hidden from the metatheory. Non functional naming relations are useful for defining meaningful naming relations that may employ more than one name of an expression of the object level, according to the extra information we want to encode.

In the next chapter, 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 metalogic languages and we will show how the corresponding encodings can be formalised. Finally, we will give the declarative semantics of the proposed language.

Chapter 3

An 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 chapter we introduce the technicalities by which names 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.

3.1 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 [79], 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.

Alphabet

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. 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.

▶ The **alphabet** of HC^+ is the union of the following disjoint sets:

- a non-empty set of predicate symbols,
- a set of function symbols,
- a non-empty set of constant symbols,
- a non-empty set of metaconstants,
- a countably infinite set of variables,
- a countably infinite set of metavariables,
- the set consisting of the operators $\uparrow$ and $\downarrow$,
- the set of the usual logical connectives and the set of punctuation symbols: ',', '(', ')','[', ']' and '$\leftarrow$'.

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_3, 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.

To ease readability we write metaconstants using the following abstract syntax. If c is a constant, a function symbol 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}.

Terms

The definition of terms in HC^+ extends the usual one to contain *name terms* 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 $[\beta_0, \beta_1, \ldots, \beta_n]$, where each β_i is the name of α_i, $0 \leq i \leq n$. Furthermore, the name of the name of $\alpha_0(\alpha_1, \ldots, \alpha_n)$ is the name term $[\gamma_0, \gamma_1, \ldots, \gamma_n]$, where each γ_i is the name of β_i, $0 \leq i \leq n$, 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.

We define name terms and terms by simultaneous induction.

▶ The set of **name terms** is defined inductively as follows:

- every metaconstant is a name term;
- every metavariable is a name term;
- every expression of the form $[\alpha_0, \alpha_1, \ldots, \alpha_n]$, $n > 0$, where α_0 is either a metavariable, a function symbol or a predicate symbol (of arity n) named m times ($m > 0$), and $\alpha_1, \ldots, \alpha_n$ are name terms, is a name term;
- for every term t, $\uparrow t$ is a name term.

Remark. We implicitly assume that whenever a metavariable X occurs in a name term of the form $[X, t_1, \ldots, t_n]$, the metavariable X can only be instantiated to a name (or a name of a name, etc.) of a function or a predicate symbol. (A more accurate definition of name terms can be given by introducing types in the language. This is the approach taken in [42], where we have introduced appropriate types for names of predicate symbols, for names of function symbols, and so on.) □

▶ The set of **terms** is defined inductively as follows:

- every constant is a term,
- every variable is a term,
- for every (n-ary) function symbol f and terms $t_1, \ldots, t_n$, $n > 0$, $f(t_1, \ldots, t_n)$ is a term,
- for every name term t, $\downarrow t$ is a term,
- every name term is a term.

Remark. We implicitly assume that in any term of the form $\downarrow t$, the name term t is not the name of any function or predicate symbol. If that were the case, then $\downarrow t$ would denote a symbol that is not a term of the language, i.e., a function or predicate symbol, respectively. Again this assumption can be cancelled by introducing appropriate types in the language. □

Remark. There are name terms that are not the name of any term. For example, let a be a constant and f a unary function symbol. Then, the name terms $[f^1, f^1]$ and $[f^2, a^1]$ do not name any term because $f(f)$ and $[f^1, a]$ are neither terms nor name terms. Our choice to allow such name terms is motivated by the fact to keep the definition of the language simple.

□

Now, we illustrate few examples of terms and name terms.

Example 3.1 Let a be a constant and f a function symbol. Let x be a variable and Y a metavariable. Then,

$$\uparrow a \qquad \uparrow\uparrow f(b,x) \qquad [Y, a^2, b^4] \qquad \uparrow[f^1, Y] \qquad \uparrow[f^2, a^3, \uparrow Y] \qquad [f^2, Y]$$

are examples of name terms, and

$$\downarrow a^1 \qquad \downarrow Y \qquad \downarrow\uparrow[f^2, a^3, \uparrow Y] \qquad \downarrow[Y, a^2, b^3]$$

are examples of terms. □

- ▶ A term of the form $f(t_1, \ldots, t_n)$, with $n > 0$, is called a **compound term**, and a name term of the form $[\alpha_0, \alpha_1, \ldots, \alpha_m]$, with $m > 0$, is called a **compound name term**.

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 $[p^1, a^1, b^1]$, where p^1 is the metaconstant that names the predicate symbol p, while the metaconstants a^1 and b^1 name the constant a and b, respectively. We may, for example, express that p is a binary predicate symbol as $binary\_pred(p^1)$. 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).

Logic Programs

We now present the definitions of definite programs, equality theories and logic programs.

- ▶ Let t_1 and t_2 be terms. Then, $t_1 = t_2$ is an **equation**. A **name equation** is an equation that contains at least one occurrence of $\uparrow$ or $\downarrow$.

  An **equality theory** is a (possibly infinite) set of equations.

- ▶ 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**.

  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 $k > 0$.[1]

- ▶ If P is a definite program and E an equality theory, then (P, E) is a **logic program**.

Given a logic program (P, E), E contains axioms characterizing $=$ (for example the usual equality interpretation of $=$ [29]), and P defines the meaning of the non-logical symbols. Observe also that, according to the definition above, atoms and equations are distinct.

NameTerm	::=	`Metaconstant`
		`Metavariable`
		`[Metaconstant,`$\mathit{NameTerm}^{+}$`]`
		`[Metavariable,`$\mathit{NameTerm}^{+}$`]`
		$\uparrow \mathit{Term}$
Term	::=	`Constant`
		`Variable`
		`Function(`Term^{+}`)`
		$\downarrow \mathit{NameTerm}$
		NameTerm
Atom	::=	`Predicate(`Term^{*}`)`
Equation	::=	*Term*`=`*Term*
DefiniteClause	::=	$\mathit{Atom} \leftarrow \mathit{Equation}^{*}, \mathit{Atom}^{*}$
DefiniteGoal	::=	$\leftarrow \mathit{Atom}^{+}$
DefiniteProgram	::=	*set of DefiniteClauses*
EqualityTheory	::=	*set of Equations*
LogicProgram	::=	`(`*DefiniteProgram*`,`*EqualityTheory*`)`

The language HC^{+}

In the figure above, α^{*} denotes a (possibly empty) sequence of α's and α^{+} denotes a non-empty sequence of α's.

What we need now is a way to formalise the relation between terms and the corresponding name terms. We do this by formalising the intended role of

[1]All our clauses and goals will be definite and so we will omit "definite" from now on.

the operators $\uparrow$ and $\downarrow$ through equational theories that are a parameter of *RCL*.

Example 3.2 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:

$$P = \{\ \mathit{starts}(x,y) \leftarrow X = \uparrow x, Y = \uparrow y, \mathit{first\_element}(X,Y)\ \}$$

$$E = \left\{ \begin{array}{ll} \uparrow\mathtt{a} = 97 & \\ \uparrow\mathtt{b} = 98 & \\ \ldots & \\ \uparrow\mathtt{z} = 122 & \\ \uparrow\mathtt{c}_1\cdots\mathtt{c}_n = [\uparrow\mathtt{c}_1,\ldots,\uparrow\mathtt{c}_n] & \text{for every constant } \mathtt{c}_1\cdots\mathtt{c}_n \end{array} \right\}$$

where $\mathit{starts}(x,y)$ holds if the constant x starts with the character y and the atom $\mathit{first\_element}(X,Y)$ holds if Y is the first element of the list X. The equality theory E formalises 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 $c_1\cdots c_n$. □

3.2 FORMALISING 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 [116]). 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 formalising encodings in this way [13, 42]. The following examples show the formalisation of some encodings appearing in the literature.

Example 3.3 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 [109].

$$\forall x \uparrow x = x. \tag{3.1}$$

This encoding, however, strongly reduces the expressive power of the meta-theory. 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 (3.1) above with the following two axioms.

$$\begin{aligned} &\text{For every constant } c, && (3.2)\\ &\uparrow c = c. &&\\ &\text{For every function symbol } f \text{ of arity } k, && (3.3)\\ &\forall x_1 \ldots \forall x_k \uparrow(f(x_1,\ldots,x_k)) = [f, \uparrow x_1, \ldots, \uparrow x_k]. && \end{aligned}$$

In (3.3) 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 [65] 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 γ.

Example 3.4 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 θ be an assignment. The Gödel number $\gamma_\theta(t)$ of t under θ is defined similarly to the Gödel numbering γ:

$$\begin{array}{rcl} \gamma_\theta(x_i) & = & n_i \\ \gamma_\theta(c_i) & = & 3^i \\ \gamma_\theta(f_j(t_1, \ldots, t_m)) & = & 3^j \times 5^{\gamma_\theta(t_1)} \times \ldots \times p_{m+2}^{\gamma_\theta(t_m)}, \end{array}$$

where each p_i is the ith prime number and the indexes of constants and function symbols are assumed to be distinct. We can formalise 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,
$\forall x_1 \ldots \forall x_k \uparrow(f_j(x_1, \ldots, x_k)) = 3^j 5^{\uparrow x_1} \times \ldots \times p_{k+2}^{\uparrow x_k}$.

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

□

The axioms below for the operators $\uparrow$ and $\downarrow$ are the basis of the formalisation 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 [42], characterize name terms and compositional names for HC^+.

▶ 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 $[X_0, X_1, \ldots, X_k]$

$$\begin{aligned}
&\forall X_0 \ldots \forall X_k \uparrow[X_0, X_1, \ldots, X_k] = [\uparrow X_0, \uparrow X_1, \ldots, \uparrow X_k].\\
&\forall x \downarrow\uparrow x = x.\\
&\forall X \uparrow\downarrow X = X.
\end{aligned}$$

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 3.3).

When an encoding has been established as being suitable for an application, its properties for sound and complete inference should be investigated. (In Chapter 6 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:

$$\begin{aligned}
&p(x) \leftarrow Y = \uparrow x, q(Y)\\
&q(a^1).
\end{aligned}$$

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 metainterpreters, whose encoding mechanism has been outlined in Example 3.3. The difficulties associated with providing them with a reasonable semantics have been discussed by Barklund et al. [14]. (See Section 3.4 for a discussion.)

3.3 E-INTERPRETATIONS

In this section we parametrize the semantics of the traditional Horn clause language with respect to an equality theory E. Whenever a semantics is defined over the Herbrand universe U, equality is interpreted by as syntactic identity. To overcome this restriction, Jaffar et al. [63] proposed the use of quotient universes. Here we adapt this technique to our context.

▶ Let U be a Herbrand universe and R 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

$$E \models s = t \quad \text{iff} \quad \lceil s \rceil_R = \lceil t \rceil_R,$$

where $\lceil s \rceil_R$ and $\lceil t \rceil_R$ denote the R-equivalence classes of the ground terms s and t, i.e., $\lceil s \rceil_R = \{x \mid x\,R\,s\}^2$. 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 every consistent Horn clause equality theory (i.e., every set of Horn clauses in which every atom is an equation) generates a finest congruence relation R_0 (the intersection of all congruence relations that are models of E). As a consequence, it holds that

$$(P, E) \models A \quad \text{iff} \quad P \models_{U/R_0} A,$$

where (P, E) is a logic program, A is a ground atom and $\models_{U/R_0}$ denotes logical implication in the context of a fixed domain and functional assignment; in this case, the domain is U/R_0 and the functional assignment is given by $f(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) = \lceil f(t_1, \ldots, t_n) \rceil$ for every n-ary function symbol f. 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, $\lceil s \rceil$ for the element in U/E assigned to the ground term s and, if p is a predicate symbol, we write $\lceil p(t_1, \ldots, t_n) \rceil$ as a shorthand for $p(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil)$.

We can now introduce the definitions of E-base, E-interpretation and E-model of a logic program (P, E).

[2] The standard notation for R-equivalence classes is $[s]_R$. We use a different notation here as we use $[\ldots]$ for compound name terms.

▶ 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.

Example 3.6 Let (P, E) be the following logic program.

$$\left(\left\{ \begin{array}{l} p(X,y) \leftarrow y = \downarrow X \\ q(a) \\ q(X) \leftarrow X = \uparrow y, q(y) \end{array} \right\}, NT \right)$$

The equality theory NT satisfies the property (see Theorem 4.10) that for every ground term s (possibly containing occurrences of $\uparrow$ and $\downarrow$), there exists a ground term t (not containing any occurrence of $\uparrow$ and $\downarrow$) such that $NT \models s = t$. Thus, we may let $\lceil s \rceil$ be t. For instance,

$$\lceil \uparrow a \rceil = a^1, \quad \lceil \uparrow\uparrow a \rceil = a^2, \quad \lceil \downarrow\downarrow a^2 \rceil = a \quad \text{and} \quad \lceil \downarrow\uparrow\uparrow a^1 \rceil = a^2.$$

Then, U/E is $\{a, a^1, \ldots, a^n, \ldots\}$ and $B_{(P,E)}$ is

$$\{p(a^n, a^m) \,|\, \text{for all}\, n > 0\, \text{and}\, m \geq 0\} \cup \{q(a^n) \,|\, \text{for all}\, n \geq 0\},$$

where we let a^0 be a. □

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

▶ 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 holds:

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

▶ 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**.

▶ 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).

Example 3.7 Let (P, E) be the logic program in Example 3.6. Then the following E-interpretation is an E-model of (P, E),

$$\{p(a^{n+1}, a^n) \mid n \geq 0\} \cup \{q(a^n) \mid n \geq 0\}.$$

□

▶ 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 $\lceil A \rceil \in I$.

The least E-model of a logic program (P, E), written as $M_{(P,E)}$, can be characterized as the least fixed point of a mapping $T_{(P,E)}$ over E-interpretations [63], written as $\mathit{lfp}(T_{(P,E)})$. Let $\mathit{ground}(P)$ be the set of all ground instances of clauses in P.

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

$$\begin{array}{ll} T_{(P,E)}(I) = \{ \lceil A \rceil : & (A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m) \in \mathit{ground}(P), \\ & E \models e_i \ \ \mathit{for}\ 1 \leq i \leq q, \\ & \lceil A_j \rceil \in I \ \ \mathit{for}\ 1 \leq j \leq m \ \}. \end{array}$$

Example 3.8 Let (P, E) be the logic program in Example 3.6. Then

$$\begin{array}{lcl} I_0 & = & \{\} \\ I_1 & = & T_{(P,E)}(I_0) = I_0 \cup \{p(a^{n+1}, a^n) \mid \text{for all}\, n \geq 0\} \cup \{q(a)\} \\ I_2 & = & T_{(P,E)}(I_1) = I_1 \cup \{q(a^1)\} \\ I_3 & = & T_{(P,E)}(I_2) = I_2 \cup \{q(a^2)\} \\ & \ldots & \\ I_{m+1} & = & T_{(P,E)}(I_m) = I_m \cup \{q(a^m)\} \\ & \ldots & \end{array}$$

We have that:

$$T_{(P,E)} \uparrow \omega = \{p(a^{n+1}, a^n) \mid n \geq 0\} \cup \{q(a^n) \mid n \geq 0\} = M_{(P,E)}.$$

□

The following result is proved by Jaffar et al. [63].

Theorem 3.9 $M_{(P,E)} = \mathit{lfp}(T_{(P,E)}) = T_{(P,E)} \uparrow \omega$.

3.4 CONCLUDING REMARKS

In a metalevel system, the choice of encoding is important, for at least two reasons. First, as we have shown with some simple examples in Section 3.2, an encoding directly determines the expressivity of the metatheory. The second reason is 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 [38, 100, 110].

In contrast, in order to give a semantic account of a metalogic programming language that employs a trivial encoding (cf. Example 3.3), two main possibilities have been considered up to now. De Schreye & Martens [41] characterize a class of programs for which the least Herbrand model semantics still hold for the vanilla metainterpreter and a limited form of amalgamation. This is the class of language-independent programs, where language independence extends both domain independence and range restrictedness. As already noted by Levi & Ramundo [78], however, the class of language independent programs is too small: it includes deductive database programs, but rules out any logic program computing partially determined data structures. Instead they point out another possibility for giving a declarative semantics to the vanilla metainterpreter and some enhanced metainterpreters, by abandoning the least Herbrand semantics in favour of the S-semantics [47]. Martens & De Schreye later extended [82] their previous results about the vanilla metainterpreters to all object programs by adopting the S-semantics, but concluded that they still do not carry over to extended metainterpreters. Besides resorting to a different semantics, that approach can be applied (as the authors of [78] recognise) to enhanced metainterpreters only when there exists a corresponding pure object-level solution. Again, this restricts the power of the language and undermines, in our view, the motivation for using metalevel systems altogether.

Our final suggestion is that of designing formalisms to be parametric with respect to encodings, so as to be able to choose the most appropriate one for the application domain at hand, while departing minimally from classical semantics.

In summary, we have defined an enhanced Horn clause language HC^+ which allows users of RCL 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 with respect to E by means of the technique of quotient universes. This is the first step (i.e., Step I) 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.

Now, we have to investigate how to extend unification so as to accommodate names, that is, given an equality theory E formalising the encoding under consideration, we have to provide (Step II) an E-unification algorithm for it. In this direction, we consider a formalisation 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. With this aim, in the next chapter we will reconsider the equality theory NT and we will present an E-unification algorithm for NT.

An E-Unification Algorithm

In this chapter, we reconsider the equality theory *NT* formalising compositional names of HC^+, and present an E-unification algorithm for *NT*. In particular, we first define a set of rules, called *NR*, suitable for computing names of HC^+ and whose underlying equality theory is *NT*. Then, we investigate which properties *NR* and *NT* enjoy. Finally, we give an E-unification algorithm, called *UN*, that uses the rules in *NR* and we explore its properties.

4.1 THE REWRITE SYSTEM NR

In this chapter, we follow the terminology of Dershowitz & Jouannaud [44]. Given a term t, we indicate with $vars(t)$ the set of variables and metavariables occurring in it. We write $t_1 \equiv t_2$ to indicate that the terms t_1 and t_2 are syntactically identical.

A *substitution*, written as $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, is a finite mapping from variables to terms and from metavariables to name terms. The variables and metavariables $x_1, \ldots, x_n$ in θ are assumed to be distinct and not occurring in any t_i, for $1 \leq i \leq n$. We also assume that for all $1 \leq i \leq n$, $x_i \not\equiv t_i$.

Substitutions operate on expressions. By an *expression* we mean a term, a sequence of atoms or equations, a clause or a set of equations. For an expression α and a substitution θ, $\alpha\theta$ stands for the result of applying θ to α which is obtained by simultaneously replacing each occurrence in α of a variable or metavariable x by the corresponding term, that is, by t if x/t is in θ.

▶ A **rewrite rule** over a set of terms is an ordered pair (l, r) of terms, written as $l \rightarrow r$, such that $vars(r) \subseteq vars(l)$. A finite set R of rewrite rules is called a **rewrite system**.

The idea of rewriting is to impose directionality on the use of equations in proofs.

The next definition introduces a rewrite system based on the equality theory *NT*. Recall that we write c^m to indicate c named m times; thus, c may be written as c^0, its name as c^1, and so on.

▶ Let **NR** be the following rewrite system. Let $n > 0$ and $m \geq 0$.

$$\boxed{\begin{array}{lrcl}
r_1: & \uparrow c^m & \rightarrow & c^{m+1} \\
r_2: & \uparrow f(x_1, \ldots, x_n) & \rightarrow & [f^1, \uparrow x_1, \ldots, \uparrow x_n] \\
r_3: & \uparrow [X_0, \ldots, X_n] & \rightarrow & [\uparrow X_0, \ldots, \uparrow X_n] \\
r_4: & \uparrow\downarrow X & \rightarrow & X \\
r_5: & \downarrow c^{m+1} & \rightarrow & c^m \\
r_6: & \downarrow [f^1, X_1, \ldots, X_n] & \rightarrow & f(\downarrow X_1, \ldots, \downarrow X_n) \\
r_7: & \downarrow [f^{m+2}, X_1, \ldots, X_n] & \rightarrow & [\downarrow f^{m+1}, \downarrow X_1, \ldots, \downarrow X_n] \\
r_8: & \downarrow\uparrow x & \rightarrow & x
\end{array}}$$

(r_5) If $m = 0$, we assume that c^{m+1} is not the name of any function or predicate symbol (see the remark on p. 17).

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 $s \underset{R}{\rightarrow} t$, if $s|_p = l\sigma$ and $t = s[r\sigma]_p$, for some rule $l{\rightarrow}r$ in R, position p in s, and substitution σ. In that case, we say that s is *reducible*. A subterm $s|_p$ at which a rewrite can take place is called *redex*. If $l{\rightarrow}r$ and $s{\rightarrow}t$ are two rewrite rules (with distinct variables) in R, p is the position of a nonvariable subterm of s, and σ is a most general unifier of $s|_p$ and l, then the equation $t\sigma = s\sigma[r\sigma]_p$ is a *critical pair* formed from those rules. A term s is *irreducible* if there is no term t such that $s \underset{R}{\rightarrow} t$. In [44] the expression *in normal form* is used as synonymous of *irreducible*. Here we define them differently and introduce the notion of *suspended form*.

▶ A term is in **normal form** if it does not contain any occurrence of $\uparrow$ or $\downarrow$.

▶ Let R be a rewrite system. A term t is in **suspended form** with respect to R if t is irreducible and t is not in normal form.

Thus, a term t is in suspended form with respect to R if t contains names that cannot be computed by any rewrite rule in R.

Example 4.1 With respect to *NR* the terms $\uparrow x$ and $\downarrow[Y, Z]$ are in suspended form as *NR* does not provide names for variables. □

Next, we characterize the class of terms in suspended form with respect to *NR*. To do this, we need the following two definitions.

▶ Let $j \geq 0$. An **s-irredex** is a term either (i) of the form $\uparrow^{j+1}x$ for some variable or metavariable x, or (ii) of the form $\downarrow^{j+1}y$ for some metavariable y.

▶ Let $j \geq 0$. A **c-irredex** is a term of the form $\downarrow^{j+1}[t_0, t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are name terms and t_0 is either a metavariable or an s-irredex.

The following proposition follows easily.

Proposition 4.2 *A term t is in suspended form with respect to NR iff t contains at least one occurrence of an s-irredex or a c-irredex.*

A *derivation* in R is any (finite or infinite) sequence $t_0 \underset{R}{\rightarrow} t_1 \underset{R}{\rightarrow} t_2 \underset{R}{\rightarrow} \ldots$ of applications of rewrite rules in R. The derivability relation $\overset{*}{\underset{\mathrm{R}}{\rightarrow}}$ is the reflexive, transitive closure of $\underset{\mathrm{R}}{\rightarrow}$. We write the inverse and the symmetric closure of a relation $\underset{\mathrm{R}}{\rightarrow}$ as $\underset{\mathrm{R}}{\leftarrow}$ and $\underset{\mathrm{R}}{\leftrightarrow}$, respectively. A rewrite system R is *terminating* if there are no infinite derivations $t_0 \underset{R}{\rightarrow} t_1 \underset{R}{\rightarrow} t_2 \underset{R}{\rightarrow} \ldots$ of terms and *convergent* (or *canonical*) if all derivations lead to a unique normal form. (Note that the notion of convergency implies termination.) R is *ground convergent* and *ground terminating* if R is convergent and terminating on ground terms, respectively.

Theorem 4.3 *The rewrite system* NR *is terminating.*

Proof. To prove that repeated applications of rewrite rules of *NR* always terminate, we define a function τ mapping terms into natural numbers as follows. Let c^m $(m \geq 0)$ be a constant or a metaconstant, and x a variable or a metavariable.

- $\tau(c^m) = \tau(x) = 1$

- $\tau(f(t_1, \ldots, t_n)) = \tau([t_0, t_1, \ldots, t_n]) = n + 2 + \tau(t_1) + \ldots + \tau(t_n)$
- $\tau(\uparrow t) = \tau(\downarrow t) = 2 \times \tau(t)$.

It is straightforward to show that $s \underset{NR}{\rightarrow} t$ implies $\tau(s) > \tau(t)$, and consequently that *NR* is terminating. ⊠

Proposition 4.4 *The rewrite system* NR *is not convergent.*

Proof. Let t be the term $\downarrow\uparrow[X, Y]$. Then, we have that $t \underset{NR}{\rightarrow} \downarrow[\uparrow X, \uparrow Y]$ using the rule r_3, and $t \underset{NR}{\rightarrow} [X, Y]$ using the rule r_8. As $[X, Y]$ and $\downarrow[\uparrow X, \uparrow Y]$ are both irreducible, the claim follows. ⊠

However, we have the positive weaker result.

Theorem 4.5 *The rewrite system NR is ground convergent.*

The proof of the statement is based on a result by Lankford [76] that is rephrased below.

First, we have to introduce the notion of confluency. Confluency says that it does not matter how much one diverges from a common ancestor, since there are paths joining to a common descendent. Thus, confluence implies the impossibility to have more than one normal form.

A binary relation $\rightarrow$ is *confluent* if the joinability relation $\overset{*}{\leftarrow} \circ \overset{*}{\rightarrow}$ is contained in the relation $\overset{*}{\rightarrow} \circ \overset{*}{\leftarrow}$. A binary relation is *ground confluent* if it is confluent on ground terms.

Note that a binary relation is ground convergent if it is ground terminating and ground confluent.

Proposition 4.6 (*Lankford, 1975*) *A ground terminating rewrite system R is ground confluent iff for all critical pairs $a = b$ formed from rewrite rules in R and for all substitutions σ such that $(a = b)\sigma$ is ground, it holds that $(a, b)\sigma \in \underset{R}{\overset{*}{\rightarrow}} \circ \underset{R}{\overset{*}{\leftarrow}}$.*

Proof. of Theorem 4.5 Ground termination of *NR* follows immediately by Theorem 4.3. Thus, to show that *NR* is ground convergent we only have to show that it is ground confluent. By Proposition 4.6 we have to prove that, for every critical pair $a = b$ and for every substitution σ such that $(a = b)\sigma$ is ground, it holds that $(a, b)\sigma \in \underset{\mathrm{NR}}{\overset{*}{\rightarrow}} \circ \underset{\mathrm{NR}}{\overset{*}{\leftarrow}}$.

The proof is simple. We establish the result only for the critical pair $[X_0, \ldots, X_n] = \downarrow [\uparrow X_0, \ldots, \uparrow X_n]$ formed from the rules r_3 and r_8. The remaining cases are similar.

Let $[t_0, t_1, \ldots, t_n] = \downarrow [\uparrow t_0, \uparrow t_1, \ldots, \uparrow t_n]$ be any ground instance of the critical pair above, where t_0 is f^{m+1} $(m \geq 0)$, for any function symbol f of arity n. By applying r_1, r_7 and r_5, we have that:

$$\begin{array}{lll} \downarrow [\uparrow f^{m+1}, \uparrow t_1, \ldots, \uparrow t_n] & \underset{\text{NR}}{\rightarrow} & \downarrow [f^{m+2}, \uparrow t_1, \ldots, \uparrow t_n] \\ & \underset{\text{NR}}{\rightarrow} & [\downarrow f^{m+2}, \downarrow\uparrow t_1, \ldots, \downarrow\uparrow t_n] \\ & \underset{\text{NR}}{\rightarrow} & [f^{m+1}, \downarrow\uparrow t_1, \ldots, \downarrow\uparrow t_n]. \end{array}$$

Finally, by applying n times r_8 to $\downarrow\uparrow t_i$, for all $1 \leq i \leq n$, we obtain $[f^{m+1}, t_1, \ldots, t_n]$. The claim follows. ⊠

▶ Given an equality theory E, a rewrite system R is **adequate for** E if

- R is terminating, and
- $s \overset{*}{\underset{\text{R}}{\leftrightarrow}} t$ iff $E \models s = t$, for any terms s and t.

Notice that the notion of adequateness does not require convergency of R, since we consider the symmetric closure of $\underset{\text{R}}{\rightarrow}$. Convergency is needed when we want to decide equality of two terms s and t via rewriting, i.e., $s \overset{*}{\underset{\text{R}}{\rightarrow}} u \overset{*}{\underset{\text{R}}{\leftarrow}} t$.

Example 4.7 Let $E = \{a = b, a = c\}$ and $R = \{a \rightarrow b, a \rightarrow c\}$. Trivially, $E \models b = c$. R is not convergent, in fact, $a \overset{*}{\underset{\text{R}}{\rightarrow}} b$ and $a \overset{*}{\underset{\text{R}}{\rightarrow}} c$, and both b and c are irreducible. By considering the symmetric closure of $\underset{\text{R}}{\rightarrow}$, we have that $b \overset{*}{\underset{\text{R}}{\leftrightarrow}} c$. □

Lemma 4.8 *Let s and t be terms. $s \underset{NR}{\leftrightarrow} t$ implies $NT \models s = t$.*

Proof. To establish the statement we have to show that for every rule $s \rightarrow t$ in *NR*, $s \underset{NR}{\leftrightarrow} t$ implies $NT \models s = t$. We establish the result only for the rule r_6, as the remaining cases are similar.

Recall that r_6 is: $\downarrow [f^1, X_1, \ldots, X_n] \rightarrow f(\downarrow X_1, \ldots, \downarrow X_n)$. In fact, with respect to *NT* we have the following equalities:

$$\begin{array}{rcll} \downarrow [f^1, X_1, \ldots, X_n] & = & f(\downarrow X_1, \ldots, \downarrow X_n) & \text{iff} \\ \uparrow\downarrow [f^1, X_1, \ldots, X_n] & = & \uparrow f(\downarrow X_1, \ldots, \downarrow X_n) & \text{iff} \\ [f^1, X_1, \ldots, X_n] & = & \uparrow f(\downarrow X_1, \ldots, \downarrow X_n) & \text{iff} \\ [f^1, X_1, \ldots, X_n] & = & [f^1, \uparrow\downarrow X_1, \ldots, \uparrow\downarrow X_n] & \text{iff} \\ [f^1, X_1, \ldots, X_n] & = & [f^1, X_1, \ldots, X_n]. & \end{array}$$

Since $NT \models \downarrow [f^1, X_1, \ldots, X_n] = f(\downarrow X_1, \ldots, \downarrow X_n)$, the claim follows. ⊠

Now, we are in the position to present the main result of this section.

Theorem 4.9 *The rewrite system* NR *is adequate for* NT.

Proof. Since by Theorem 4.3 *NR* is terminating, it suffices to prove that for any terms s and t, $s \underset{NR}{\overset{*}{\leftrightarrow}} t$ iff $NT \models s = t$.

(i) $s \underset{NR}{\overset{*}{\leftrightarrow}} t$ implies $NT \models s = t$.

Assume that $s \underset{NR}{\overset{*}{\leftrightarrow}} t$. Then, there exists a sequence of terms $s_0, \ldots, s_n$ such that $s_0 \equiv s$, $s_n \equiv t$ and $s_i \underset{NR}{\leftrightarrow} s_{i+1}$, for $0 \leq i < n$. By induction on i, for $i \leq n$, and by Lemma 4.8, the statement (i) follows.

(ii) $NT \models s = t$ implies $s \underset{NR}{\overset{*}{\leftrightarrow}} t$.

Given an equality theory E, define the relation $\underset{\mathrm{E}}{\leftrightarrow}$ over terms as follows: $a \underset{\mathrm{E}}{\leftrightarrow} b$ iff $a \equiv u[l\sigma]_p$ and $b \equiv u[r\sigma]_p$ for some term u, position p in u, equation $l = r$ (or $r = l$) in E, and substitution σ.

Assume that $NT \models s = t$. By Birkhoff's completeness theorem [18], it holds that $s \underset{NT}{\overset{*}{\leftrightarrow}} t$. Thus, there exists a sequence of terms $s_0, \ldots, s_n$ such that $s_0 \equiv s$, $s_n \equiv t$ and $s_i \underset{NT}{\leftrightarrow} s_{i+1}$, for $0 \leq i < n$. From the definition of *NT* and *NR* it follows immediately that $a \underset{NT}{\leftrightarrow} b$ implies $a \underset{NR}{\leftrightarrow} b$. By induction on i, for $i \leq n$, and by noting that $a \underset{NT}{\leftrightarrow} b$ implies $a \underset{NR}{\leftrightarrow} b$, the statement (ii) follows. ⊠

4.2 FORMAL PROPERTIES OF NT

In this section we prove some properties of the equality theory *NT*. We use the following definition.

- ▶ Let E be an equality theory. E is **sufficiently complete** if for every ground term s there exists a ground term t in normal form such that $E \models s = t$.

Sufficiently complete equality theories enjoy the property that, for every ground term s, there exists a term t in normal form provably equal to s. For example, s may be a name term containing names to be computed and t may be a term provably equal to s with its names computed.

Theorem 4.10 *NT is sufficiently complete.*

The proof of the statement uses a lexicographic path ordering over terms (defined below). We treat the square brackets of name terms as if they were function symbols, i.e., a name term of the form $[t_0, \ldots, t_n]$ is treated like $[\,](t_o, \ldots, t_n)$.

▶ A **precedence relation** is any total ordering on the signature $\mathcal{F}$ of the language, i.e., the set of constants, metaconstants and function symbols.

▶ Let $\succ_{\mathcal{F}}$ be a precedence relation on $\mathcal{F}$, and let $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ be terms. The **lexicographic path ordering** $\succ$ induced by $\succ_{\mathcal{F}}$ is the ordering on terms defined recursively as follows.

$s \succ t$ if one of the following holds:

- $s_i \succeq t$, for some i, with $1 \leq i \leq m$; or
- $f \succ_{\mathcal{F}} g$ and $s \succ t_j$, for all j, with $1 \leq j \leq n$; or
- $f \equiv g$ (and so $n = m$), $(s_1, \ldots, s_m) \succ^{lex} (t_1, \ldots, t_m)$ and $s \succ t_j$, for all j, with $1 \leq j \leq m$,

where $(s_1, \ldots, s_m) \succ^{lex} (t_1, \ldots, t_m)$ if there is $j \leq m$ such that $s_j \succ t_j$ and, for all $i < j$, we have $s_i = t_i$.

Note that a lexicographic path ordering is well-founded, and is total on ground terms since the precedence relation $\succ_{\mathcal{F}}$ is total on the signature.

Proof. of Theorem 4.10 Consider any precedence relation $\succ_{\mathcal{F}}$ that satisfies the following conditions:

$$\begin{array}{ll} \uparrow \succ_{\mathcal{F}} c^m & \downarrow \succ_{\mathcal{F}} c^m \\ \uparrow \succ_{\mathcal{F}} f & \downarrow \succ_{\mathcal{F}} f \\ \uparrow \succ_{\mathcal{F}} [\,] & \downarrow \succ_{\mathcal{F}} [\,] \end{array}$$

for every constant or metaconstant c^m ($m \geq 0$) and for every function symbol f different from $\uparrow$ and $\downarrow$. The proof of the statement is by induction on the lexicographic path ordering $\succ$ induced by $\succ_{\mathcal{F}}$.

Base Case. Immediate (the term is in normal form).

Inductive Step. We establish the result only for name terms of the form $\uparrow f(t_1, \ldots, t_n)$. The remaining cases are similar.

By the definition of *NT*, the name term $\uparrow f(t_1, \ldots, t_n)$ is equal to the name term $[f^1, \uparrow t_1, \ldots, \uparrow t_n]$. Since $\uparrow f(t_1, \ldots, t_n) \succ [f^1, \uparrow t_1, \ldots, \uparrow t_n]$, the statement follows by the induction hypothesis. ⊠

▶ Let E be an equality theory split into a set E_n of name equations and a set $E_o = E - E_n$ of other equations. E is **consistent** if, for all ground terms s and t in normal form, it holds that:

$$E \models s = t \quad \text{iff} \quad E_o \models s = t.$$

Consistency guarantees that whenever we extend an equality theory E_0 (defining equality among terms in normal form) with an equality theory E_n of name equations, equality between terms in normal form is unchanged. For instance, if $E_0 = \{\}$ and $E_n = \{\uparrow a = a^1, \uparrow a = b^1\}$, then $E = E_0 \cup E_n$ is not consistent, in fact, $E_0 \not\models a^1 = b^1$ and $E \models a^1 = b^1$.

The following result shows that the equality theory *NT* is consistent. In this case, we have that $E = NT$ and, consequently, $E_0 = \{\}$ and $E_n = NT$. This means that, for any ground terms s and t in normal form, $NT \models s = t$ iff $s \equiv t$.

Corollary 4.11 *NT is consistent.*

Proof. Immediate by Theorem 4.5 and Theorem 4.9. ⊠

Consistency of *NT* is an important property because it allows us to treat equality between terms in normal form as syntactic equality. In turn, this means that if we want to specify an E-unification algorithm for *NT*, we may consider extending algorithms for syntactic unification. For instance, we may consider Martelli and Montanari's algorithm for unifying terms in normal form, and extend it to deal with the operations of quoting and unquoting.

4.3 THE E-UNIFICATION ALGORITHM UN

In the context of equational logic programming, unification algorithms are usually expressed in terms of rewrite rules based on sets of equations rather than on substitutions. Unification algorithms expressed as rewrite systems on sets of equations have been defined, e.g, by Huet [62], Robinson [98] and Martelli & Montanari [81].

In order to take into account names of the language HC^+, we define an E-unification algorithm, called *UN*, that uses the rewrite system *NR*. To do this, we need some definitions.

▶ A **binding** is an equation either of the form:

– $x = t$ if x is a variable, t is a term and x does not occur in t, or

  - $y = s$ if y is a metavariable, s is a name term and y does not occur in s.

- ▶ A **Herbrand assignment** $H = \{x_1 = t_1, \ldots, x_k = t_k\}$ is a set (possibly, empty) of bindings such that:
  - the variables and metavariables x_i, with $1 \leq i \leq k$, are distinct,
  - no x_i is in any t_j, for every $1 \leq i, j \leq k$, 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., equations with names that still have to be computed. Every Herbrand assignment $H = \{x_1 = t_1, \ldots, x_k = t_k\}$ determines a substitution $\{x_1/t_1, \ldots, x_k/t_k\}$, which we indicate with $\widehat{H}$.

- ▶ A **transformation rule**, written as $\Rightarrow$, is an ordered pair of tuples 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. A **transformation system** is a finite set of transformation rules.

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 names that have to be computed.

A transformation system is *convergent* (or *canonical*) if all sequences of transformations lead to a unique form.

Remark. Rewrite rules and transformation rules are similar concepts. We have introduced two distinct names for them to disambiguate the case where the rule operates over terms and where it operates over tuples of sets of equations. □

The following transformation system extends Martelli & Montanari's transformation system [81] to take into account metavariables and name equations. (A slightly different version of it was first introduced in [42].) This transformation system uses the rewrite system *NR* for compositional names. Below we write an equation yet to be solved as $s \overset{?}{=} t$ and we assume the predicate $\overset{?}{=}$ to be symmetric.

Delete:
$\langle H, F, S \cup \{t \overset{?}{=} t\}\rangle \Rightarrow \langle H, F, S\rangle$

Decompose:
$\langle H, F, S \cup \{f(t_1, \ldots, t_n) \overset{?}{=} f(s_1, \ldots, s_n)\}\rangle \Rightarrow \langle H, F, S \cup \{t_1 \overset{?}{=} s_1, \ldots, t_n \overset{?}{=} s_n\}\rangle$
for any n-ary function symbol f (including $\uparrow$ and $\downarrow$).

SplitCnt:
$\langle H, F, S \cup \{[t_0, \ldots, t_n] \overset{?}{=} [s_0, \ldots, s_n]\}\rangle \Rightarrow \langle H, F, S \cup \{t_0 \overset{?}{=} s_0, \ldots, t_n \overset{?}{=} s_n\}\rangle$
with $n > 0$.

Eliminate:
$\langle H, F, S \cup \{x \overset{?}{=} t\}\rangle \Rightarrow \langle H\theta \cup \{x = t\}, F\theta, S\theta\rangle$
if $x = t$ is a binding and t is in normal form. θ is $\{x/t\}$.

Freeze:
$\langle H, F, S \cup \{x \overset{?}{=} t\}\rangle \Rightarrow \langle H, F\theta \cup \{x = t\}, S\theta\rangle$
if $x = t$ is a binding and t is in suspended form. θ is $\{x/t\}$.

Unfreeze:
$\langle H, F \cup \{x = t\}, S\rangle \Rightarrow \langle H, F, S \cup \{x \overset{?}{=} t\}\rangle$
if t is reducible.

Rewrite:
$\langle H, F, S \cup \{t \overset{?}{=} s\}\rangle \Rightarrow \langle H, F, S \cup \{t_1 \overset{?}{=} s\}\rangle$
if $t \underset{NR}{\rightarrow} t_1$.

Martelli & Montanari's system is extended here with four rules. The first new rule, *splitCnt*, decomposes compound name terms. The rules *freeze* and *unfreeze* move name equations from S to F, and vice versa. If $x = t$ is a binding and t is in suspended form, that is, t contains names that cannot be computed, then *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 *eliminate* or *freeze*. If t becomes reducible, *unfreeze* moves $x = t$ back to S, where it can subsequently be reduced. The last rule, *rewrite*, allows us to compute names with respect to the rewrite system *NR*. If a name equation $t \overset{?}{=} s$ contains a term t reducible to some term t_1, i.e., $t \underset{NR}{\rightarrow} t_1$, then *rewrite* replaces t in $t \overset{?}{=} s$ with t_1.

Remark. A well-known technique to solve equations in the presence of an equality theory uses the 'narrowing' transformation on terms. A term s *narrows* to a term t, via a substitution σ, if t is $s\sigma[r\sigma]_p$, for some position

p of a nonvariable subterm of s, rule $l \to r$ in R and a most general unifier σ of $s|_p$ and l. The transformation system above uses rewriting rather than narrowing since the rewrite system *NR* contains axiom schemata. Thus, there are terms that narrow to an infinite number of terms. Consider, for instance, the name term $\uparrow x$. It narrows to c^1 via the substitution $\{x/c\}$, to c^2 via $\{x/c^1\}$, and so on. By employing rewriting instead, $\uparrow x$ cannot be rewritten. □

Example 4.12 Given $S = \{f(X, Y, \uparrow X) = f(\uparrow a, \downarrow Z, Z)\}$, the tuple

$$\langle\{\}, \{\}, \{f(X, Y, \uparrow X) = f(\uparrow a, \downarrow Z, Z)\}\rangle$$

can be transformed using the transformation rules above in the following steps:

$$\begin{array}{lcll}
(0) & \langle\{\}, \{\}, \{f(X, Y, \uparrow X) \stackrel{?}{=} f(\uparrow a, \downarrow Z, Z)\}\rangle & & \\
(1) & \Rightarrow & \langle\{\}, \{\}, \{X \stackrel{?}{=} \uparrow a, Y \stackrel{?}{=} \downarrow Z, \uparrow X \stackrel{?}{=} Z\}\rangle & \text{(by decompose)} \\
(2) & \Rightarrow & \langle\{\}, \{Y = \downarrow Z\}, \{X \stackrel{?}{=} \uparrow a, \uparrow X \stackrel{?}{=} Z\}\rangle & \text{(by freeze)} \\
(3) & \Rightarrow & \langle\{\}, \{Y = \downarrow Z\}, \{X \stackrel{?}{=} a^1, \uparrow X \stackrel{?}{=} Z\}\rangle & \text{(by rewrite)} \\
(4) & \Rightarrow & \langle\{X = a^1\}, \{Y = \downarrow Z\}, \{\uparrow a^1 \stackrel{?}{=} Z\}\rangle & \text{(by eliminate)} \\
(5) & \Rightarrow & \langle\{X = a^1\}, \{Y = \downarrow Z\}, \{a^2 \stackrel{?}{=} Z\}\rangle & \text{(by rewrite)} \\
(6) & \Rightarrow & \langle\{X = a^1, Z = a^2\}, \{Y = \downarrow a^2\}, \{\}\rangle & \text{(by eliminate)} \\
(7) & \Rightarrow & \langle\{X = a^1, Z = a^2\}, \{\}, \{Y \stackrel{?}{=} \downarrow a^2\}\rangle & \text{(by unfreeze)} \\
(8) & \Rightarrow & \langle\{X = a^1, Z = a^2\}, \{\}, \{Y \stackrel{?}{=} a^1\}\rangle & \text{(by rewrite)} \\
(9) & \Rightarrow & \langle\{X = a^1, Z = a^2, Y = a^1\}, \{\}, \{\}\rangle & \text{(by eliminate)}
\end{array}$$

In the first step, *decompose* is applied. In step 2, *freeze* is applied to the binding $Y \stackrel{?}{=} \downarrow Z$. The tuple in 4 is obtained by applying *rewrite* and *eliminate* to $X \stackrel{?}{=} \uparrow a$. By using *rewrite* and *eliminate* on the name equation $\uparrow a^1 \stackrel{?}{=} Z$, we obtain the tuple in 6. Then, since the name term $\downarrow a^2$ in $Y = \downarrow a^2$ is reducible, the name equation is moved back (step 7) and rewritten to $Y \stackrel{?}{=} a^1$. Finally, the binding $Y \stackrel{?}{=} a^1$ is eliminated. □

In general, given a set of equations S, it is desirable to have a transformation system that allows us to transform S into a set of bindings, possibly in some "solved" form (i.e., a form for which an obvious solution always exists). The transformation system above is not able to do so. For example, consider the set $S = \{\uparrow x = c^1\}$. None of the transformation rules is applicable to S and thus the equation $\uparrow x = c^1$ cannot be transformed into a binding (e.g., $x = \downarrow c^1$). With this in mind, we introduce 5 transformation rules that allow us to simplify name equations.

Below we write *nonvar*(t) to indicate that the term t is neither a variable nor a metavariable, and *functor*(t) to indicate the functor of the term t. Let $i \geq 0$ and $j \geq 0$.

S1: $\langle H, F, S \cup \{\uparrow^{i+1} x \overset{?}{=} t\}\rangle \Rightarrow \langle H, F, S \cup \{x \overset{?}{=} \downarrow^{i+1} t\}\rangle$
if $\uparrow^{i+1} x$ is an s-irredex, nonvar(t) and functor$(t) \notin \{\uparrow, \downarrow\}$.

S2: $\langle H, F, S \cup \{\downarrow^{i+1} x \overset{?}{=} t\}\rangle \Rightarrow \langle H, F, S \cup \{x \overset{?}{=} \uparrow^{i+1} t\}\rangle$
if $\downarrow^{i+1} x$ is an s-irredex, nonvar(t) and functor$(t) \notin \{\uparrow, \downarrow\}$.

S3: $\langle H, F, S \cup \{\downarrow^{i+1} [t_0, \ldots, t_n] \overset{?}{=} t\}\rangle \Rightarrow \langle H, F, S \cup \{[t_0, \ldots, t_n] \overset{?}{=} \uparrow^{i+1} t\}\rangle$
if $\downarrow^{i+1} [t_0, \ldots, t_n]$ is a c-irredex, nonvar(t) and functor$(t) \notin \{\uparrow, \downarrow\}$.

S4: $\langle H, F, S \cup \{\uparrow^{i+1} x \overset{?}{=} c\}\rangle \Rightarrow \langle H, F, S \cup \{x \overset{?}{=} \downarrow^{i+1} c\}\rangle$
if $\uparrow^{i+1} x$ is an s-irredex and c is a c-irredex.

S5: $\langle H, F, S \cup \{\downarrow^{i+1} x \overset{?}{=} \uparrow^{j+1} y\}\rangle \Rightarrow \langle H, F, S \cup \{x \overset{?}{=} \uparrow^{i+j+2} y\}\rangle$
if $\downarrow^{i+1} x$ and $\uparrow^{j+1} y$ are s-irredexes.

The following examples illustrate the use of the transformation rules above.

Example 4.13 Given $S = \{\uparrow X = [Y, a^4], \uparrow Z = \downarrow X\}$, the tuple

$$\langle \{\}, \{\}, \{\uparrow X = [Y, a^4], \uparrow Z = \downarrow X\}\rangle$$

can be rewritten using the transformation rules above in the following steps:

(0)	$\langle \{\}, \{\}, \{\uparrow X \overset{?}{=} [Y, a^4], \uparrow Z \overset{?}{=} \downarrow X\}\rangle$		
(1)	$\Rightarrow$	$\langle \{\}, \{\}, \{\uparrow X \overset{?}{=} [Y, a^4], \uparrow\uparrow Z \overset{?}{=} X\}\rangle$	(by *S5*)
(2)	$\Rightarrow$	$\langle \{\}, \{X = \uparrow\uparrow Z\}, \{\uparrow\uparrow\uparrow Z \overset{?}{=} [Y, a^4]\}\rangle$	(by freeze)
(3)	$\Rightarrow$	$\langle \{\}, \{X = \uparrow\uparrow Z\}, \{Z \overset{?}{=} \downarrow\downarrow\downarrow [Y, a^4]\}\rangle$	(by *S1*)
(4)	$\Rightarrow$	$\langle \{\}, \{X = \uparrow\uparrow\downarrow\downarrow\downarrow [Y, a^4], Z = \downarrow\downarrow\downarrow [Y, a^4]\}, \{\}\rangle$	(by freeze)
(5)	$\Rightarrow$	$\langle \{\}, \{Z = \downarrow\downarrow\downarrow [Y, a^4]\}, \{X \overset{?}{=} \uparrow\uparrow\downarrow\downarrow\downarrow [Y, a^4]\}\rangle$	(by unfreeze)
(6)	$\Rightarrow$	$\langle \{\}, \{Z = \downarrow\downarrow\downarrow [Y, a^4]\}, \{X \overset{?}{=} \uparrow\downarrow\downarrow [Y, a^4]\}\rangle$	(by rewrite)
(7)	$\Rightarrow$	$\langle \{\}, \{Z = \downarrow\downarrow\downarrow [Y, a^4]\}, \{X \overset{?}{=} \downarrow [Y, a^4]\}\rangle$	(by rewrite)
(8)	$\Rightarrow$	$\langle \{\}, \{Z = \downarrow\downarrow\downarrow [Y, a^4], X = \downarrow [Y, a^4]\}, \{\}\rangle$	(by freeze)

In the first step, *S5* is applied. In step 2, *freeze* is applied to the binding $\uparrow\uparrow Z \overset{?}{=} X$. Then, by applying *S1* and *freeze* to $\uparrow\uparrow\uparrow Z \overset{?}{=} [Y, a^4]$, the tuple in step 4 is obtained. Finally, since the name term $\uparrow\uparrow\downarrow\downarrow\downarrow[Y, a^4]$ is reducible, the equation is moved back, reduced and frozen again. □

Example 4.14 Given $S = \{\downarrow X = f(y), \downarrow X = g(z)\}$, the tuple

$$\langle \{\}, \{\}, \{\downarrow X \overset{?}{=} f(y), \downarrow X \overset{?}{=} g(z)\}\rangle$$

can be rewritten using the transformation rules above in the following steps:

$$\begin{array}{lcll}
(0) & \langle \{\}, \{\}, \{\downarrow X \overset{?}{=} f(y), \downarrow X \overset{?}{=} g(z)\}\rangle & & \\
(1) & \Rightarrow & \langle \{\}, \{\}, \{X \overset{?}{=} \uparrow f(y), \downarrow X \overset{?}{=} g(z)\}\rangle & \text{(by } S2\text{)} \\
(2) & \Rightarrow & \langle \{\}, \{\}, \{X \overset{?}{=} [f^1, \uparrow y], \downarrow X \overset{?}{=} g(z)\}\rangle & \text{(by rewrite)} \\
(3) & \Rightarrow & \langle \{\}, \{X = [f^1, \uparrow y]\}, \{\downarrow [f^1, \uparrow y] \overset{?}{=} g(z)\}\rangle & \text{(by freeze)} \\
(4) & \Rightarrow & \langle \{\}, \{X = [f^1, \uparrow y]\}, \{f(\downarrow\uparrow y) \overset{?}{=} g(z)\}\rangle & \text{(by rewrite)} \\
(5) & \Rightarrow & \langle \{\}, \{X = [f^1, \uparrow y]\}, \{f(y) \overset{?}{=} g(z)\}\rangle & \text{(by rewrite)}
\end{array}$$

In the first step, *S2* is applied. In step 2, *rewrite* is applied to $X \overset{?}{=} \uparrow f(y)$. The tuple in step 3 is obtained by applying *freeze* to the binding $X \overset{?}{=} [f^1, \uparrow y]$. Finally, by applying twice *rewrite* to $\downarrow[f^1, \uparrow y] \overset{?}{=} g(z)$, we obtain the tuple in step 5 to which no rule is applicable. Note that the third part of the tuple in 5 is non empty. This fact indicates the absence of a solution for S. □

Recall that we indicate a Herbrand assignment and its corresponding substitution with H and $\widehat{H}$, respectively.

▶ Let E be a sufficiently complete equality theory. An **E-solution** of an equation $s = t$ is a Herbrand assignment H such that $E \models s\widehat{H} = t\widehat{H}$. An E-solution of a set S of equations is a Herbrand assignment that is an E-solution of every equation in S. In that case, S is called **E-satisfiable**.

Observe that in the definition of E-solution we can restrict our attention to Herbrand assignments because E is sufficiently complete.

▶ Let E be an equality theory sufficiently complete. A transformation rule is **sound for** E if, whenever $\langle H, F, S\rangle \Rightarrow \langle H_1, F_1, S_1\rangle$, the sets $H \cup F \cup S$ and $H_1 \cup F_1 \cup S_1$ admit the same set of E-solutions.

As *NT* is sufficiently complete, we have the following result.

Proposition 4.15 *The transformation rules delete, decompose, splitCnt, eliminate, freeze, unfreeze, rewrite, S1, S2, S3, S4 and S5 are sound for NT.*

In the following, given a set S of equations, we call a Herbrand assignment H an *NT-solution* of S if H is an E-solution of S with respect to the equality theory *NT*, i.e., $NT \models S\widehat{H}$. Recall that we use $\equiv$ to indicate syntactic identity.

The proof of Proposition 4.15 uses the following result, the proof of which immediately follows from the definition of composition of substitutions [6, 79].

Proposition 4.16 *Let $x = t$ be a binding and H a Herbrand assignment. If $x\widehat{H} \equiv t\widehat{H}$, then $\{x/t\}\widehat{H}$ and $\widehat{H}$ are identical.*

Proof. of Proposition 4.15 The proof of the statement is trivial for the rules: *delete*, *decompose*, *splitCnt* and *unfreeze*.

For the rules *S1*, *S2*, *S3*, *S4* and *S5*, the statement follows by observing that those rules only redirect the arrows $\uparrow$ and $\downarrow$ of equations in S and that, for any name equations of the form $\downarrow t = s$ and $\uparrow t = s$, we have that:

(i) $\downarrow t = s$ implies (by substitutivity)
$\uparrow\downarrow t = \uparrow s$ implies (by the last axiom in NT)
$t = \uparrow s$;

(ii) $\uparrow t = s$ implies
$\downarrow\uparrow t = \downarrow s$ implies
$t = \downarrow s$.

We establish the result for *rewrite*:

$$\langle H, F, S \cup \{t = s\}\rangle \Rightarrow \langle H, F, S \cup \{t_1 = s\}\rangle, \quad \text{if} \quad t \underset{NR}{\rightarrow} t_1.$$

We have to prove that, for every Herbrand assignment H_1, H_1 is an *NT*-solution of $H \cup F \cup S \cup \{t = s\}$ iff H_1 is an *NT*-solution of $H \cup F \cup S \cup \{t_1 = s\}$.

Assume that $t \underset{NR}{\rightarrow} t_1$. Since by Theorem 4.9 *NR* is adequate for *NT*, we have that $NT \models t = t_1$. It follows from $NT \models t = t_1$ that $NT \models t\widehat{H_1} = t_1\widehat{H_1}$. The claim follows.

Now, we establish the result for *eliminate*:

$$\langle H, F, S \cup \{x = t\}\rangle \Rightarrow \langle H\theta \cup \{x = t\}, F\theta, S\theta\rangle, \quad \text{with} \quad \theta = \{x/t\},$$

where $x = t$ is a binding and t is in normal form. We have to show that, for every Herbrand assignment H_1, H_1 is an *NT*-solution of $H \cup F \cup S \cup \{x = t\}$ iff H_1 is an *NT*-solution of $H\theta \cup \{x = t\} \cup F\theta \cup S\theta$.

This result follows by noting that $x\widehat{H_1} \equiv t\widehat{H_1}$ (because H_1 is an *NT*-solution of $x = t$ and t is in normal form) and by Proposition 4.16 we have that $\theta\widehat{H_1}$ and $\widehat{H_1}$ are identical.

Finally, we establish the result for *freeze*:

$$\langle H, F, S \cup \{x = t\}\rangle \Rightarrow \langle H, F\theta \cup \{x = t\}, S\theta\rangle, \quad \text{with} \quad \theta = \{x/t\},$$

where $x = t$ is a binding and t is in suspended form. Observe first that every *NT*-solution H_1 of $x = t$ must contain a binding for x, since (by the definition of binding) $x \not\equiv t$. Suppose $H_1 = \{x = s_0, y_1 = s_1, \ldots, y_m = s_m\}$. Then, $\theta\widehat{H_1} = \{x/t\widehat{H_1}, y_1/s_1, \ldots, y_m/s_m\}$.

We have to show that, for every Herbrand assignment H_1, H_1 is an *NT*-solution of $H \cup F \cup S \cup \{x = t\}$ iff H_1 is an *NT*-solution of $H \cup F\theta \cup \{x = t\} \cup S\theta$.

The result follows immediately by noting that $NT \models s_0 = t\widehat{H_1}$ because H_1 is an *NT*-solution of $x = t$ and $x\widehat{H_1} \equiv s_0$. ⊠

Next, we show that, by applying the transformation rules until none is applicable, results in a tuple that is "almost" solvable. That is, we obtain a tuple that in general is in not solvable, but whose solvability can be easily checked.

▶ A tuple $\langle H, F, S\rangle$ is in **quasi-solved form** iff it satisfies the following conditions:

1. $S = \{\}$,
2. H is a Herbrand assignment, say $H = \{x_1 = t_1, \ldots, x_n = t_n\}$,
3. no x_i in H occurs in F,
4. F is a set of bindings, say $F = \{y_1 = s_1, \ldots, y_m = s_m\}$,
5. the variables and metavariables y_i in F are distinct,
6. no y_i occurs in any s_j, and
7. every s_j is in suspended form.

Given the equality theory *NT*, we call a set S of equations *NT-satisfiable* if S is E-satisfiable with respect to *NT*.

The following statement shows that by applying the transformation rules presented above to any *NT*-satisfiable set of equations results in a tuple in quasi-solved form. However, the converse does not hold, that is, there are tuples in quasi-solved form that are not *NT*-satisfiable. Observe that no transformation rule is applicable to a tuple in quasi-solved form.

Proposition 4.17 *Let S_0 be a finite set of equations. Assume that starting with $\langle \{\}, \{\}, S_0 \rangle$ and applying the transformation rules delete, decompose, splitCnt, eliminate, freeze, unfreeze, rewrite, S1, S2, S3, S4 and S5 until none is applicable results in $\langle H, F, S \rangle$. If S_0 is NT-satisfiable, then $\langle H, F, S \rangle$ is in quasi-solved form.*

Proof. (Outline) Assume that S_0 is *NT*-satisfiable. We have to show that conditions 1–7 of the definition of quasi-solved form hold for $\langle H, F, S \rangle$.

- Condition 1 ($S = \{\}$). Condition 1 follows by noting that S_0 is *NT*-satisfiable and every transformation rule above is sound. Thus, if S contained any equation, then one of the transformation rule above (except *unfreeze*) would be applicable.
- Conditions 2–6. Observe first that the only transformation rules that modify H and F are *eliminate*, *freeze* and *unfreeze*.

  We show that the conditions 2–6 hold for the tuple $\langle H, F, S \rangle$ by induction on the number of applications of *eliminate*, *freeze* and *unfreeze*.

  Base Case. Trivial for $\langle \{\}, \{\}, S_0 \rangle$.

  Inductive Step. Suppose that $\langle H_i, F_i, S_i \rangle \Rightarrow \langle H_{i+1}, F_{i+1}, S_{i+1} \rangle$ by applying one of the rules *eliminate*, *freeze* or *unfreeze*. By the induction hypothesis H_i and F_i satisfy the conditions 2–6. We want to show that also H_{i+1} and F_{i+1} satisfy those conditions. We distinguish among three cases depending on which transformation rule is used.

  – *Eliminate*. Let $H_i = \{x_1 = t_1, \ldots, x_n = t_n\}$ and $F_i = \{y_1 = s_1, \ldots, y_m = s_m\}$. Suppose that $x = t$ is a binding, t is in normal form and $\theta = \{x/t\}$ is the substitution used by *eliminate*. Then, H_{i+1} is $H_i\theta \cup \{x = t\}$, F_{i+1} is $F_i\theta$ and S_{i+1} is $(S_i - \{x = t\})\theta$.

  Note that no x_j (with $1 \leq j \leq n$) occur anywhere else in the tuple $\langle H_i, F_i, S_i \rangle$ by the definitions of binding and *eliminate*. Thus, $x \not\equiv x_j$, for every $1 \leq j \leq n$, and after the application of θ to both H_i, F_i and S_i also x occurs only once in $\langle H_{i+1}, F_{i+1}, S_{i+1} \rangle$.

  Note also that no y_l (with $1 \leq l \leq m$) occur anywhere else in the sets F_i and S_i by the definitions of binding and *freeze*. Thus, $x \not\equiv y_l$, for every $1 \leq l \leq m$.

  Condition 2 holds for H_{i+1} because H_i does not contain any binding of the form $x = s$, for any term s; $x = t$ is a binding and x does not occur in $H_i\theta$.

Since by the induction hypothesis no x_j (with $1 \leq j \leq n$) occur in F_i and x does not occur in $F_i\theta$, condition 3 holds.

Conditions 4 and 5 hold by noting that $x \not\equiv y_l$ implies that y_l is not instantiated by θ, for all $1 \leq l \leq m$. Thus, each equation $(y_l = s_l)\theta$ in $F_i\theta$ is a binding. The variables and metavariables in $y_1, \ldots, y_m$ are distinct by the induction hypothesis.

As no y_l occurs neither in $s_1, \ldots, s_m$ (by the induction hypothesis), nor in t, condition 6 holds for $F_i\theta$.

– *Freeze*: similar to the previous case.

– *Unfreeze*: obvious.

- Condition 7. Since *unfreeze* is not applicable to $\langle H, F, S\rangle$, condition 7 for quasi-solved form holds for $\langle H, F, S\rangle$.

The claim follows. $\boxtimes$

Given a tuple $\langle H, F, \{\}\rangle$ in quasi-solved form, the set H is trivially *NT*-satisfiable, while it is not always the case for F. If F contains, for instance, a name equation of the form $x = \downarrow\downarrow[Z, t_1, \ldots, t_n]$, then we have to make sure that every name term t_i can be unnamed twice. For example, there may be a name term t_i of the form $[g^1, a^1]$. Such a name term can be unnamed only once. Thus, if $\langle H, F, \{\}\rangle$ is in quasi-solved form, $H \cup F$ is not necessarily *NT*-satisfiable.

The next example shows that metavariables in name equations in F may interact.

Example 4.18 Consider the two name equations $y_1 = \downarrow[X, [g^1, a^1]]$ and $y_2 = \downarrow[\downarrow X, [g^2, a^2]]$. They are (separately) both *NT*-satisfiable, for instance, with *NT*-solutions $\{y_1 = f(g(a)), X = f^1\}$ and $\{y_2 = f(g(a)), X = f^2\}$, respectively. However, the set $F = \{y_1 = \downarrow[X, [g^1, a^1]], y_2 = \downarrow[\downarrow X, [g^2, a^2]]\}$ containing both is not *NT*-satisfiable, since the first equation requires X to be instantiated to a name of a function symbol, while the second equation requires X to be instantiated to a name of a name of a function symbol. $\square$

Thus, given a tuple $\langle H, F, \{\}\rangle$ in quasi-solved form, we need to test *NT*-satisfiability of $H \cup F$. This is what the *NT*-satisfiability-test does. We give here a "brute force" algorithm with the only aim to show that the *NT*-satisfiability problem is decidable.

An Algorithm for NT-satisfiability

```
NT-satisfiability-test ($\langle H, F, S\rangle$)

1.  if not quasi-solved-form($\langle H, F, S\rangle$)
2.      then return $\langle\{\}, \{\}, \{false\}\rangle$
3.  $k$ := max-number-$\downarrow$-in-equations($F$) +1
4.  $V$ := set-of-metavar-occurring-in-1°-argument-of-c-irredexes($F$)
5.  if instantiate-and-check-NT-satisfiability($k$,$V$,$F$)
6.      then return $\langle H, F, S\rangle$
7.  return $\langle\{\}, \{\}, \{false\}\rangle$
```

The algorithm performs the following steps. First, it checks whether the input tuple $\langle H, F, S\rangle$ is in quasi-solved form. If this is not the case, then the algorithm returns the tuple $\langle\{\}, \{\}, \{false\}\rangle$ to indicate the absence of any *NT*-solution (in fact, by Proposition 4.17, $\langle H, F, S\rangle$ is not *NT*-satisfiable). Otherwise ($\langle H, F, S\rangle$ is in quasi-solved form), the algorithm counts (step 3) the number of occurrences of $\downarrow$ in every equation in F and assigns the highest number increased by 1 to k. Then, it assigns (step 4) the set of all metavariables occurring in the first argument of c-irredexes in name equations in F to V. Finally, the algorithm returns either the input tuple (step 6) if the test instantiate-and-check-*NT*-satisfiability(k,V,F) (defined below) succeeds, or the tuple $\langle\{\}, \{\}, \{false\}\rangle$ (step 7).

instantiate-and-check-*NT*-satisfiability(k,V,F)

The test instantiate-and-check-*NT*-satisfiability is a subroutine of the kind "generate and test". It consists of two parts. The first part generates Herbrand assignments H_i for the metavariables in V, and the second part tests whether $F\widehat{H_i}$ is *NT*-satisfiable. If such a H_i exists, then the subroutine terminates successfully, otherwise it terminates unsuccessfully.

In the following, let $V = \{X_1, \ldots, X_n\}$. The first part of the subroutine generates Herbrand assignments of the form $H_i = \{X_1 = t_1, \ldots, X_n = t_n\}$, where every t_j $(1 \leq j \leq n)$ is a (selected) function symbol named l_j times, with $1 \leq l_j \leq k$. That is, t_j may be f^{l_j} where f is the selected function symbol. (Note that, since the metavariables $X_1, \ldots, X_n$ are distinct, it is not relevant which function symbol is selected (apart from its arity). Clearly, if we have n metavariables in V, then, in the worst case, we have to generate all combinations of the numbers $l_1, \ldots, l_n$, with $1 \leq l_j \leq k$.) Since $\downarrow$ occurs at most $k-1$ times in any name equation in F, it suffices to consider name terms f^{l_j}, with $l_j \leq k$ (see the remark below).

The first part of the subroutine obviously terminates because the set V of metavariables is finite and every l_j is constrained by $1 \leq l_j \leq k$.

For every Herbrand assignment H_i generated, the second part of the subroutine tests *NT*-satisfiability for $F\widehat{H_i}$ through the following steps:

```
1.  $F1$ := apply-rewrite-rules-of-$NR$-to($F\widehat{H_i}$)
2.  if set-of-bindings($F1$)
3.      then return true
4.  return false
```

First, it applies the rewrite rules in *NR* (step 1) to terms of equations in $F\widehat{H_i}$ as many times as possible. Then, it checks (step 2) whether every equation in $F1$ is a binding. If this is the case, the subroutine returns true meaning that $F\widehat{H_i}$ is *NT*-satisfiable. Otherwise, $F\widehat{H_i}$ is not *NT*-satisfiable and the subroutine reports this fact by returning (step 4) false.

Observe that $F1$ may contain equations that are not bindings. For instance, if F contains the name equation $X = \downarrow[Y, a^2]$ and $H_1 = \{Y = f^1\}$ is an Herbrand assignment generated, then $F1$ will contain $X = f(a^1)$. Since X is a metavariable and $f(a^1)$ is a term, but not a name term, the equation $X = f(a^1)$ is not a binding. Thus, it is needed to check (step 2) whether every equation in $F1$ is a binding.

Observe also that there may exist equations in $F1$ containing entities of the form $\downarrow t$, where t is a term, but not a name term. Thus, $\downarrow t$ is not a term and the equation containing it, is not a binding.

The second part of the subroutine terminates because the rewrite system *NR* is terminating by Theorem 4.3.

Remark. We have to increase the highest number of occurrences of $\downarrow$ in any equation of F by 1 because we may want to have a name term after having instantiated and rewritten c-irredexes. Reconsider the set $F = \{X = \downarrow[Y, a^2]\}$. Obviously, F is *NT*-satisfiable. If we would not have increased that number by 1, then the only Herbrand assignment generated would have been $H_1 = \{Y = f^1\}$, which is not an *NT*-solution. (Since X is a metavariable, X needs to be bound to a name term.) Instead, by considering $H_2 = \{Y = f^2\}$, the equation $X = \downarrow[f^2, a^2]$ in $F\widehat{H_2}$ is rewritten to $X = [f^1, a^1]$, which is a binding. Considering Herbrand assignments $H_l = \{Y = f^l\}$, with $l \geq 2$, is superfluous. $\square$

Example 4.19 Let $\langle H, F, \{\}\rangle$ be a tuple in quasi-solved form, where $F = \{y_1 = \downarrow[X, [g^1, a^1]], y_2 = \downarrow[\downarrow X, [g^2, a^2]]\}$. F is not NT-satisfiable (see Example 4.18). NT-satisfiability-test detects this fact as follows. First, it calculates the highest number of occurrences of $\downarrow$ in equations of F, and it assigns such a number increased by 1 to k. In this case, $k = 3$. Then, it computes the set V of metavariables. In this case, $V = \{X\}$. Finally, it performs the test instantiate-and-check-NT-satisfiability(k,V,F). This subroutine first generates an Herbrand assignment H_i, instantiates the first argument of the c-irredexes in F by H_i, rewrites them and performs the test for bindings. Since the Herbrand assignments generated are: $H_1 = \{X = f^1\}$, $H_2 = \{X = f^2\}$ and $H_3 = \{X = f^3\}$, and the test set-of-bindings fails in all three cases, the subroutine returns false, and in turn NT-satisfiability-test returns the tuple $\langle\{\}, \{\}, \{false\}\rangle$.

For $i = 2$, for instance, the sets $F\widehat{H_2}$ and F1 are:

$$F\widehat{H_2} = \{y_1 = \downarrow[f^2, [g^1, a^1]], y_2 = \downarrow[\downarrow f^2, [g^2, a^2]]\}$$

$$F1 = \{y_1 = [f^1, g(a)], y_2 = f([g^1, a^1])\}$$

Since $[f^1, g(a)]$ is neither a name term nor a term, the first equation in F1 is not a binding. Thus, the test set-of-bindings($F1$) fails. □

The next result shows the properties enjoyed by NT-satisfiability-test.

Proposition 4.20 *The following properties hold.*

(*i*) *NT-satisfiability-test is terminating.*

(*ii*) *If a tuple $\langle H, F, S\rangle$ is in quasi-solved form and $H \cup F$ is NT-satisfiable, then NT-satisfiability-test($\langle H, F, S\rangle$) = $\langle H, F, S\rangle$.*

(*iii*) *If a tuple $\langle H, F, S\rangle$ is not in quasi-solved form or $H \cup F$ is not NT-satisfiable, then NT-satisfiability-test($\langle H, F, S\rangle$) = $\langle\{\}, \{\}, \{false\}\rangle$.*

The proof of the statement uses the following two lemmas whose proofs easily follows from the definition of quasi-solved form.

Lemma 4.21 *Let $\langle H, F, S\rangle$ be a tuple in quasi-solved form. If F does not contain any occurrence of c-irredexes, then $H \cup F$ is NT-satisfiable.*

Lemma 4.22 *Let $\langle H, F, S\rangle$ be a tuple in quasi-solved form. Let V be the set of all metavariables occurring in the first argument of a c-irredex in some name equation of F. Let k be the highest number of occurrences of $\downarrow$ in the name equations of F increased by 1. If $H \cup F$ is NT-satisfiable, then there exists an NT-solution H_1 of $H \cup F$ such that (i) H_1 contains a binding $X_j = t_j$, for every metavariable X_j in V, and (ii) t_j is a name term of the form f^{l_j}, with $l_j \leq k$, for some function symbol f.*

Proof. of Proposition 4.20

(i) Trivial.

(ii) Assume that $\langle H, F, S\rangle$ is in quasi-solved form (so $S = \{\}$) and $H \cup F$ is NT-satisfiable. We must prove that:

$$NT\text{-satisfiability-test}(\langle H, F, S\rangle) = \langle H, F, S\rangle.$$

Since $\langle H, F, S\rangle$ is in quasi-solved form, NT-satisfiability-test computes k and V, and then executes the test:

$$\text{instantiate-and-check-}NT\text{-satisfiability}(k,V,F).$$

To prove statement (ii), we have to show that this test succeeds.

In fact, the first part of the test generates Herbrand assignments H_i binding all the metavariables X_j in V with name terms t_j of the form f^{l_j}, with $l_j \leq k$, for some (selected) function symbol f. By Lemma 4.22, there exists an Herbrand assignment $H_{\bar{i}}$ (among those that are generated) such that $(H \cup F)\widehat{H_{\bar{i}}}$ is NT-satisfiable. Consider now $H_{\bar{i}}$.

The second part of the test rewrites all c-irredexes in $F\widehat{H_{\bar{i}}}$ (note that every c-irredex in $F\widehat{H_{\bar{i}}}$ has its first argument ground) and assigns the resulting set to $F1$. By Theorem 4.9, $F\widehat{H_{\bar{i}}}$ and $F1$ admit the same set of NT-solutions. Thus, $H \cup F1$ is NT-satisfiable and, consequently, every equation in $F1$ is a binding.

instantiate-and-check-NT-satisfiability(k,V,F) succeeds (step 3, p. 47) because every equation in $F1$ is a binding, and, consequently, we have that NT-satisfiability-test$(\langle H, F, S\rangle) = \langle H, F, S\rangle$.

(iii) If $\langle H, F, S\rangle$ is not in quasi-solved form, then NT-satisfiability-test returns $\langle\{\}, \{\}, \{\mathit{false}\}\rangle$ (step 2).

Assume that $\langle H, F, S\rangle$ is in quasi-solved form and $H \cup F$ is not NT-satisfiable. Then, by Lemma 4.21 F must contain at least one occurrence of a c-irredex that is the "source" of NT-unsatisfiability.

In this case, NT-satisfiability-test computes k and V, and performs the test instantiate-and-check-NT-satisfiability(k,V,F).

This test fails for the following reason. For no Herbrand assignment H_i generated to bind the metavariables in V, the set $F\widehat{H_i}$ is NT-satisfiable (by assumption). Thus, after rewriting c-irredexes in $F\widehat{H_i}$, the resulting set $F1$ is not NT-satisfiable by Theorem 4.9.

Observe that, by the form of $F\widehat{H_i}$, $F1$ has to contain equations that are not bindings in order to be not NT-satisfiable. This fact is detected by the test set-of-bindings($F1$) (step 2, p. 47). ⊠

Now, we are in the position to introduce an E-unification algorithm whose underlying equality theory is NT.

The E-unification Algorithm UN

▶ The E-unification algorithm **UN** is any procedure that:

1. takes a finite set S_0 of equations and uses the transformation rules *delete*, *decompose*, *splitCnt*, *eliminate*, *freeze*, *unfreeze*, *rewrite*, *S1*, *S2*, *S3*, *S4* and *S5* to generate sequences of tuples from $\langle \{\}, \{\}, S_0 \rangle$. If using the rules above until none is applicable results in $\langle H, F, S \rangle$, then
2. the procedure applies the test for NT-satisfiability and returns the tuple NT-satisfiability-test($\langle H, F, S \rangle$).

Starting with $\langle \{\}, \{\}, S_0 \rangle$ and using the rules above until none is applicable results either in $\langle \{\}, \{\}, \{\mathit{false}\} \rangle$ iff S_0 is not NT-satisfiable, or in a tuple $\langle H, F, \{\} \rangle$ in quasi-solved form such that $H \cup F$ is NT-satisfiable (see Corollary 4.26 below). The former situation signifies failure, while in the latter case, a most general unifier can be extracted from H.

We write $\langle H, F, S \rangle \underset{\text{UN}}{\rightsquigarrow} \langle H', F', S' \rangle$ to indicate that the E-unification algorithm UN takes as input the tuple $\langle H, F, S \rangle$ and returns the tuple $\langle H', F', S' \rangle$.

Example 4.23 Given $S_0 = \{f(X, Y, \uparrow X) = f(\uparrow a, \downarrow Z, Z)\}$, UN first rewrites the tuple

$$\langle \{\}, \{\}, \{f(X, Y, \uparrow X) = f(\uparrow a, \downarrow Z, Z)\} \rangle$$

to the tuple (see Example 4.12):

$$\langle \{X = a^1, Z = a^2, Y = a^1\}, \{\}, \{\} \rangle.$$

Then, UN applies NT-satisfiability-test to $\langle \{X = a^1, Z = a^2, Y = a^1\}, \{\}, \{\} \rangle$ and returns the tuple itself. □

4.4 FORMAL PROPERTIES OF UN

We require that any E-unification algorithm satisfies some properties that will play a central role in the proofs of the soundness and completeness of SLD$^{\mathcal{R}}$-resolution defined in Chapter 5. We investigate here which properties are enjoyed by *UN*.

The first result proves termination.

Theorem 4.24 *The E-unification algorithm UN is terminating.*

The proof of the statement uses two functions, ϕ and γ, mapping sets of equations into natural numbers. They are defined as follows.

Given a set of equations S, let

- $\gamma_1(S)$ be the number of distinct variables and metavariables in S,
- $\gamma_2(S)$ be the number of variables and metavariables x with only one occurrence in S such that x occurs in an equation (in S) of the form $x = t$, for any t.

$\gamma(S)$ is defined as:

$$\gamma(S) = \gamma_1(S) - \gamma_2(S).$$

Given a set of equations S, let

- $\phi_1(S)$ be the number of occurrences of variables and metavariables in S,
- $\phi_2(S)$ be the number of equations $x = t$ in S, where x is either a variable or a metavariable, and t is any term;
- $\phi_3(S)$ be the number of equations $[t_0, t_1, \ldots, t_n] = t$ in S, where:
  - t_0 is either a metavariable or an s-irredex,
  - t_1,…,t_n are name terms, and
  - t is a name term either of the form $[s_0, \ldots, s_m]$, or $\uparrow^{j+1}[s_0, \ldots, s_m]$ or $\uparrow^{j+1} f(s_0, \ldots, s_m)$, for terms $s_0, \ldots, s_m$ and $j \geq 0$;
- $\phi_4(S)$ be the number of equations $s = t$, where s is an s-irredex and the term t is neither a variable nor a metavariable.

$\phi(S)$ is defined as:

$$\phi(S) = 2 \times \phi_1(S) - [2 \times \phi_2(S) + \phi_3(S) + \phi_4(S)].$$

Proof. of Theorem 4.24 To prove that repeated applications of transformation rules of *UN* always terminate, we define a function Φ mapping a tuple $\langle H, F, S\rangle$ into a tuple of natural numbers $(n_1, n_2, n_3, n_4, n_5, n_6, n_7)$, as follows. The first number, n_1, is the number of distinct variables and metavariables in F and S. n_2 is $\gamma(F \cup S)$. n_3 is the number of occurrences of variables and metavariables in F and S. n_4 is the number of equations in F of the form $x = t$ and t is reducible. n_5 is $\phi(S)$. n_6 is $\sum_{e \in S} \tau(e)$, where $\tau(a = b)$ is $\tau(a) + \tau(b)$, for every equation $a = b$ in S, and τ over terms is defined as in the proof of Theorem 4.3. Finally, n_7 is the number of equations in S.

Let $\succ$ be the lexicographic ordering on N^7. With the above ordering, N^7 becomes a well-founded set, i.e., a set where no infinite decreasing sequence exists. Thus, if we prove that any transformation rule of *UN* transforms a tuple $\langle H, F, S\rangle$ into a tuple $\langle H', F', S'\rangle$ such that $\Phi(\langle H, F, S\rangle) \succ \Phi(\langle H', F', S'\rangle)$, we have proved termination.

In fact, *delete*, *decompose* and *splitCnt* decrease n_6; *eliminate* decreases n_1; *freeze* decreases n_7 and if it increases n_3, n_4, n_5 or n_6 then it decreases n_2; *unfreeze* decreases n_4; *rewrite* decreases n_6; finally, the rules *S1*, *S2*, *S3*, *S4* and *S5* decrease n_5.

Since by Proposition 4.20 *NT*-satisfiability-test is terminating, the claim follows. ⊠

We need to show now that whenever *UN* takes as input a tuple $\langle H, F, S\rangle$ and produces a tuple $\langle H', F', S'\rangle$, that is, $\langle H, F, S\rangle \underset{\text{UN}}{\rightsquigarrow} \langle H', F', S'\rangle$, the sets $H \cup F \cup S$ and $H' \cup F' \cup S'$ admit the same set of *NT*-solutions, i.e., *UN* is sound for *NT*.

Theorem 4.25 *The E-unification algorithm UN is sound for NT.*

Proof. Immediate by Proposition 4.15 and Proposition 4.20. ⊠

Corollary 4.26 *Let S_0 be a finite set of equations. The E-unification algorithm UN rewrites $\langle\{\}, \{\}, S_0\rangle$ either*

(*i*) *to $\langle\{\}, \{\}, \{false\}\rangle$, if S_0 is not NT-satisfiable, or*

(*ii*) *to a tuple $\langle H, F, \{\}\rangle$ in quasi-solved form, if S_0 is NT-satisfiable. Furthermore, S_0 and $H \cup F$ admit the same set of NT-solutions.*

Proof. Suppose that starting with $\langle\{\}, \{\}, S_0\rangle$ and by applying the transformation rules until none is applicable (i.e., by executing part 1 of *UN*) results in a tuple $\langle H_1, F_1, S_1\rangle$.

(*i*) Suppose that S_0 is not *NT*-satisfiable. We have to prove that *UN* returns $\langle \{\}, \{\}, \{\mathit{false}\}\rangle$. We distinguish between two cases depending on whether or not $\langle H_1, F_1, S_1\rangle$ is in quasi-solved form.

Case 1. $\langle H_1, F_1, S_1\rangle$ is not in quasi-solved form.
Then, *NT*-satisfiability-test($\langle H_1, F_1, S_1\rangle$) returns $\langle \{\}, \{\}, \{\mathit{false}\}\rangle$ by Proposition 4.20 (case (*iii*)).

Case 2. $\langle H_1, F_1, S_1\rangle$ is in quasi-solved form (so $S_1 = \{\}$).
Then, $H_1 \cup F_1$ is not *NT*-satisfiable because S_0 is not *NT*-satisfiable and S_0 and $H_1 \cup F_1$ admit the same set of *NT*-solutions by Proposition 4.15. Finally, *NT*-satisfiability-test($\langle H_1, F_1, S_1\rangle$) returns $\langle \{\}, \{\}, \{\mathit{false}\}\rangle$ by Proposition 4.20 (case (*iii*)).

(*ii*) Assume that S_0 is *NT*-satisfiable. Then, $\langle H_1, F_1, S_1\rangle$ is in quasi-solved form (so $S_1 = \{\}$) by Proposition 4.17. Since, by Proposition 4.15, S_0 and $H_1 \cup F_1$ admit the same set of *NT*-solutions, $H_1 \cup F_1$ is *NT*-satisfiable. $\langle H_1, F_1, S_1\rangle$ being in quasi-solved form and $H_1 \cup F_1$ being *NT*-satisfiable, *NT*-satisfiability-test($\langle H_1, F_1, S_1\rangle$) returns $\langle H_1, F_1, S_1\rangle$ by Proposition 4.20 (case (*ii*)).

Let $\langle H, F, \{\}\rangle$ be $\langle H_1, F_1, S_1\rangle$; the claim (*ii*) follows.

⊠

Like the rewrite system *NR*, *UN* does not converge, that is, not all sequences of transformations lead to a unique form. There are two reasons for non convergence: (*i*) because *UN* uses the rewrite rules of *NR* and *NR* is not convergent, (*ii*) by the transformation rules of *UN* themselves (as the proof of the next proposition shows).

Proposition 4.27 *The E-unification algorithm UN does not converge.*

Proof. Let $S_0 = \{X = Y, Y = \uparrow Z\}$. *UN* can rewrite S_0 in two different ways: either, by applying *eliminate* and *freeze*,

$$(i) \qquad \begin{array}{ll} \langle \{\}, \{\}, \{X \overset{?}{=} Y, Y \overset{?}{=} \uparrow Z\}\rangle & \\ \Rightarrow & \langle \{X = Y\}, \{\}, \{Y \overset{?}{=} \uparrow Z\}\rangle \\ \Rightarrow & \langle \{X = Y\}, \{Y = \uparrow Z\}, \{\}\rangle \end{array}$$

or, by applying *freeze* twice,

$$(ii) \qquad \begin{array}{ll} \langle \{\}, \{\}, \{X \overset{?}{=} Y, Y \overset{?}{=} \uparrow Z\}\rangle & \\ \Rightarrow & \langle \{\}, \{Y = \uparrow Z\}, \{X \overset{?}{=} \uparrow Z\}\rangle \\ \Rightarrow & \langle \{\}, \{Y = \uparrow Z, X = \uparrow Z\}, \{\}\rangle. \end{array}$$

Since the tuples $\langle \{X = Y\}, \{Y = \uparrow Z\}, \{\} \rangle$ and $\langle \{\}, \{Y = \uparrow Z, X = \uparrow Z\}, \{\} \rangle$ are in quasi-solved form, no transformation rules are applicable to them. ⊠

This chapter has presented an example of an E-unification algorithm that is based on an equality theory formalising compositional names of HC^+. The formal properties that such an algorithm has to satisfy when integrated into a computational framework have been investigated. This is the second step for specifying the deductive system, *DS*, under consideration. Then, the user of *RCL* has to represent (Step III) the axioms defining *DS* in the metalanguage HC^+. Now, as the last step (i.e., Step IV) for specifying a *DS*, we have to provide the user the possibility of defining the inference rules of *DS* and performing deductions in *DS*. That is the topic of the next chapter.

Chapter 5

Reflection Principles and Reflective Semantics

The last step when specifying a deductive system *DS* is representing its inference rules. In *RCL*, this is accomplished by means of reflection principles. The novelty of the approach is precisely that newly defined inference rules are immediately "executable".

The user of *RCL* is required to express an inference rule R as a function $\mathcal{R}$, called a reflection principle, from clauses, which constitute the antecedent of the rule, to sets of clauses, which constitute the consequent. Then, given a basic theory T consisting of a set of initial axioms A (enhanced Horn clauses) and of its deductive closure, and given a reflection principle $\mathcal{R}$, 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 $\mathcal{R}$. Consequently, the model-theoretic and fixpoint semantics of T under $\mathcal{R}$ are obtained as the model-theoretic and fixpoint semantics of T'. However, *RCL* 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*).

5.1 THE CONCEPT OF REFLECTION PRINCIPLE

The idea of reflection in logic dates back to work by Feferman [48]. 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 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 language the inference rules of a deductive system *DS* that a user of *RCL* 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. We choose instead to represent 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 may or may not formally follow from the validity of the given program, but rather follows $\underline{\textit{conceptually}}$, 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 fixpoint semantics of the given theory under the new inference rule coincide with the corresponding semantics [64] 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 clause (selected in the program) of any resolution step (see the definition of SLD$^{\mathcal{R}}$-resolution).

In this and the following sections, we present a formalisation of our concept of reflection which should constitute a simple way of understanding reflective programs as well as description of how reflection allows one to uniformly treat different application areas. The applications of reflection that we have previously studied (and reported in detail in previous papers [34, 38, 39]) are instances of the new formalisation. 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 reflection principles suitable for capturing 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 the previous use of reflection rules in logic programming, such as in [19]. 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, *different* from the set of theorems that are provable from the basic theory alone. This capability allows one to model several forms of reasoning within a single formal framework.

We define reflection principles as follows.

- ▶ 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 5.1 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 *demo* defined over representations of propositions in T itself, such that *demo* holds for all those representations for which the corresponding propositions are provable. This can be formalised as:

$$\frac{\alpha_i}{\mathit{demo}(\ulcorner \alpha_i \urcorner)}$$

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

$$\mathcal{R}(\alpha_i) = \{\mathit{demo}(\ulcorner \alpha_i \urcorner) \leftarrow \alpha_i\}.$$

Assume that T contains the following initial set of axioms:

$$A = \{\alpha_1, \alpha_2, \alpha_3 \leftarrow \mathit{demo}(\ulcorner \alpha_2 \urcorner)\}.$$

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

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

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

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

Notice that several reflection principles can co-exist in the same framework. This is the case in the application outlined in Chapter 7. □

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 $(P \cup \mathcal{R}(P), E)$ 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 some 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.

Example 5.2 Let C be any definite clause of the form $H \leftarrow B$ and $\mathcal{R}$ be defined as:

$$\mathcal{R}(C) = \{demo(X) \leftarrow X = \uparrow H, B\}.$$

Then, $\Delta_{\mathcal{R}}$ could be the following Prolog program:

$:-\ op(1200, xfx, \text{'}\leftarrow\text{'}).$
$:-\ op(1000, xfy, \text{','}).$
$:-\ op(700, xfx, \text{'='}).$
$:-\ op(500, fx, \text{'}\uparrow\text{'}).$

$eq(X, X).$
$deltaR(C, L){:-}\ eq(C, (H \leftarrow B)), !, eq(L, [(demo(X) \leftarrow X = \uparrow H, B)]).$
$deltaR(H, L){:-}\ eq(L, [(demo(X) \leftarrow X = \uparrow H)]).$

□

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.

The following sections give a model-theoretic and functional characterization of logic programs and present an extension to SLD-resolution that takes reflection principles into consideration.

5.2 REFLECTIVE E-MODELS AND FIXED POINT SEMANTICS

To be able to use the results of this chapter, we shall assume from now on that we consider equality theories that are sufficiently complete.

We use the following definitions.

- ▶ Let $\mathcal{R}$ 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 $\mathcal{R}$) if and only if I E-satisfies $(P \cup \mathcal{R}(P), E)$.

- ▶ Let (P, E) be a logic program. (P, E) is **reflectively E-satisfiable** if and only if there exists an E-interpretation I of (P, E) such that I reflectively E-satisfies (P, E). Otherwise, (P, E) is **reflectively E-unsatisfiable**.

- ▶ 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 [64], 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 the least reflective E-model of (P, E), indicated in the following as $M^{\mathcal{R}}_{(P,E)}$. It entails the consequences of (P, E), the additional consequences drawn by means of the reflection axioms, and the further consequences obtained from both. $M^{\mathcal{R}}_{(P,E)}$ 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 program and the reflection axioms.

Example 5.3 Let (P, E) be the following logic program:

$$\left(\left\{ \begin{array}{l} q \leftarrow demo(p^1) \\ p \end{array} \right\}, NT \right).$$

Given a definite clause C of the form $H \leftarrow B$, we define a reflection principle $\mathcal{R}$ as:

$$\mathcal{R}(C) = \{demo(H^1) \leftarrow B\}.$$

Then, we have that

$$\mathcal{R}(P) = \{demo(q^1) \leftarrow demo(p^1), demo(p^1)\},$$

and the reflective E-model of (P, E) is:

$$M^{\mathcal{R}}_{(P,E)} = \{p, demo(p^1), q, demo(q^1)\}.$$

$\square$

▶ 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 $\lceil A \rceil \in I$.

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 5.4 *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 $A_1 \wedge \ldots \wedge A_k$ is a reflective logical E-consequence of (P, E).*

Proof. Suppose that $(P, E) \cup \{\leftarrow A_1, \ldots, A_k\}$ 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 $(P, E) \cup \{\leftarrow A_1, \ldots, A_k\}$ is reflectively E-unsatisfiable, I cannot be a reflective E-model of $\neg(A_1 \wedge \ldots \wedge A_k)$. 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 $A_1 \wedge \ldots \wedge A_k$ is a reflective logical E-consequence of (P, E).

Conversely, suppose that $A_1 \wedge \ldots \wedge A_k$ is a reflective logical E-consequence of (P, E). Let I be a reflective E-model of (P, E). Then I is also a reflective E-model of $A_1 \wedge \ldots \wedge A_k$. Hence, I is not a reflective E-model of $\neg(A_1 \wedge \ldots \wedge A_k)$. Consequently, $(P, E) \cup \{\leftarrow A_1, \ldots, A_k\}$ is reflectively E-unsatisfiable. ⊠

The least reflective E-model of a logic program (P, E) can be characterized as the least fixed point of a mapping $T^{\mathcal{R}}_{(P,E)}$ that extends $T_{(P,E)}$ [63]. The extension is based on the presence of reflection axioms.

▶ Let $\mathcal{R}$ be a reflection principle, (P, E) a logic program and I an E-interpretation of (P, E). $T^{\mathcal{R}}_{(P,E)}$ is defined as follows:

$$\begin{aligned} T^{\mathcal{R}}_{(P,E)}(I) = \{ \lceil A \rceil :\ & (A \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m) \in ground(P \cup \mathcal{R}(P)), \\ & E \models e_i \ \ for\ 1 \leq i \leq q, \\ & \lceil A_j \rceil \in I \ \ for\ 1 \leq j \leq m \ \}. \end{aligned}$$

Example 5.5 Let (P, E) be the logic program in Example 5.3. Then, we have that:

$$\begin{aligned}
I_0 &= \{\} \\
I_1 &= T^{\mathcal{R}}_{(P,E)}(I_0) = I_0 \cup \{p, \mathit{demo}(p^1)\} \\
I_2 &= T^{\mathcal{R}}_{(P,E)}(I_1) = I_1 \cup \{q, \mathit{demo}(q^1)\} \\
I_3 &= T^{\mathcal{R}}_{(P,E)}(I_2) = I_2 = M^{\mathcal{R}}_{(P,E)}.
\end{aligned}$$

□

Next we give a fixed point characterization of the least reflective E-model of a logic program. We do this by first proving two useful propositions. We use the following (standard) definitions.

Let S be a set with a partial order $\sqsubseteq$ and X a subset of S. $a \in S$ is an *upper bound* of X if $x \sqsubseteq a$, for all $x \in X$. a is the *least upper bound* of X, written as $\mathit{lub}(X)$, if a is an upper bound of X and $a \sqsubseteq a'$, for all upper bounds a' of X. Similarly, $b \in S$ is a *lower bound* of X if $b \sqsubseteq x$, for all $x \in X$. b is the *greatest lower bound* of X, written as $\mathit{glb}(X)$, if b is a lower bound of X and $b' \sqsubseteq b$, for all lower bounds b' of X.

A partially ordered set $(L, \sqsubseteq)$ is a *complete lattice* if $\mathit{lub}(Y)$ and $\mathit{glb}(Y)$ exist for every subset Y of L. A subset Y of L is *directed* if every finite subset of Y has an upper bound in Y.

Let $(L, \sqsubseteq)$ be a complete lattice and $T : L \to L$ a mapping. T is *monotonic* if $T(x) \sqsubseteq T(y)$, whenever $x \sqsubseteq y$. T is *continuous* if $T(\mathit{lub}(Y)) = \mathit{lub}(T(Y))$, for every directed subset Y of L. $a \in L$ is a *fixed point* of T if $T(a) = a$, and a is a *prefixed point* of T if $T(a) \sqsubseteq a$. If a is a fixed point of T and $a \sqsubseteq a'$, for all fixed points a' of T, then a is the *least fixed point* of T. Similarly, we define the *least prefixed point* of T.

Proposition 5.6 *Let $\mathcal{R}$ be a reflection principle and (P, E) a logic program. The mapping $T^{\mathcal{R}}_{(P,E)}$ is continuous.*

Proof. To prove the statement, we have to show that, for every directed subset X of $2^{B_{(P,E)}}$, it holds that $T^{\mathcal{R}}_{(P,E)}(\mathit{lub}(X)) = \mathit{lub}(T^{\mathcal{R}}_{(P,E)}(X))$. Let X be a directed subset of $2^{B_{(P,E)}}$.

Notice first that $\{\lceil A_1 \rceil, \ldots, \lceil A_m \rceil\} \subseteq \mathit{lub}(X)$ iff $\{\lceil A_1 \rceil, \ldots, \lceil A_m \rceil\} \subseteq I$, for some $I \in X$. Now we have that:

$\lceil A \rceil \in T^{\mathcal{R}}_{(P,E)}(\mathit{lub}(X))$

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

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

iff $\lceil A \rceil \in \mathit{lub}(T^{\mathcal{R}}_{(P,E)}(X))$.

The statement follows. ⊠

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

Proposition 5.7 *Let $\mathcal{R}$ be a reflection principle, (P, E) a logic program and I an E-interpretation of (P, E). I is a reflective E-model of (P, E) if and only if $T^{\mathcal{R}}_{(P,E)}(I) \subseteq I$.*

Proof. I is a reflective E-model for (P, E);

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

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

As the class of E-interpretations forms a complete lattice under the inclusion order [63], $T^{\mathcal{R}}_{(P,E)}$ is continuous over this class, and the class of reflective E-models is given by $\{I \mid T^{\mathcal{R}}_{(P,E)}(I) \subseteq I\}$, we can provide (similarly to the standard case, van Emden & Kowalski [114]) a fixed point characterization of the least reflective E-model of a logic program (P, E).

Theorem 5.8 *Let $\mathcal{R}$ be a reflection principle and (P, E) a logic program. Then, $M^{\mathcal{R}}_{(P,E)} = \mathit{lfp}(T^{\mathcal{R}}_{(P,E)}) = T^{\mathcal{R}}_{(P,E)} \uparrow \omega$.*

The proof of the theorem is simple and uses the following two well-known facts.

Lemma 5.9 (*Kleene*) *Let $(L, \sqsubseteq)$ be a complete lattice and $T : L \rightarrow L$ be continuous. Then, $\mathit{lfp}(T) = T \uparrow \omega$.*

Theorem 5.10 (*Fixed point theorem*) (*Knaster & Tarski*)
Let $(L, \sqsubseteq)$ be a complete lattice and $T : L \rightarrow L$ be monotonic. Define:

$$m = \mathit{glb}(\{x \in L : T(x) \sqsubseteq x\}).$$

Then, m is a fixed point of T and the least prefixed point of T.

Clearly, m is the least fixed point of T.

Proof. of Theorem 5.8 It suffices to apply Propositions 5.6 and 5.7, Lemma 5.9 and Theorem 5.10. ⊠

Next we introduce the definitions of answer and correct answer.

▶ 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 name equations such that $\langle H, F, \{\}\rangle$ is in quasi-solved form and $H \cup F$ is E-satisfiable.

▶ 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 $\forall((A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'})$ is a reflective logical E-consequence of (P, E), for every E-solution H' of F.

Theorem 5.11 *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 $(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is ground, then the following are equivalent:*

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

(b) *$(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is true w.r.t. every reflective E-model of (P, E),*

(c) *$(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is true w.r.t. the least reflective E-model of (P, E).*

Proof.

$(a) \Rightarrow (c)$
Immediate from the definitions of correct answer and reflective logical E-consequence.

$(c) \Rightarrow (b) \Rightarrow (a)$
$(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is true w.r.t. the least reflective E-model of (P, E)
implies $(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is true w.r.t. all reflective E-models of (P, E)
implies $\neg(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is false w.r.t. all reflective E-models of (P, E)
implies $(P, E) \cup \{\leftarrow (A_1, \ldots, A_k)\widehat{H}\widehat{H'}\}$ has no reflective E-models
implies $(P, E) \cup \{\leftarrow (A_1, \ldots, A_k)\widehat{H}\widehat{H'}\}$ is reflectively E-unsatisfiable
implies $(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is a reflective logical E-consequence of (P, E) by
Proposition 5.4 since $(A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'}$ is ground
implies $\langle H, F\rangle$ is a correct answer because H' is an E-solution of F. ⊠

5.3 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, e.g., Clark [30]). 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. In this process unification can be seen as a rewrite system that takes a set of equations to an equivalent Herbrand assignment [81].

The assumption that unification rewrites the whole set of equations to a Herbrand assignment can be relaxed. We define a state as follows.

▶ A **state** is a triple $\langle M, H, F \rangle$, where M is a set of atoms, H a Herbrand assignment and F a set of name equations such that the tuple $\langle H, F, \{\} \rangle$ is in quasi-solved form.

▶ Let $\langle H, F, \{\} \rangle$ be a tuple in quasi-solved form. Let (P, E) be a logic program and G a definite goal of the form $\leftarrow A_1, \ldots, A_m$. Then, an **initial state** for refuting $(P, E) \cup \{G\}$ starting with (H, F) is the state $\langle \{A_1, \ldots, A_m\}, H, F \rangle$. A **success state** is a state of the form $\langle \{\}, H', F' \rangle$, where $H' \cup F'$ is E-satisfiable.

Let E be an equality theory and $e_1, \ldots, e_n$ equations. Let $\langle H, F, \{\} \rangle$ be a tuple in quasi-solved form. We write $\rightsquigarrow$ to indicate any E-unification algorithm for E that is:

1. terminating,
2. sound for E, and
3. takes $\langle H, F, \{e_1, \ldots, e_n\} \rangle$ either
   - to $\langle \{\}, \{\}, \{\mathit{false}\} \rangle$ if $H \cup F \cup \{e_1, \ldots, e_n\}$ is not E-satisfiable, or
   - to a tuple $\langle H', F', \{\} \rangle$ in quasi-solved form if $H \cup F \cup \{e_1, \ldots, e_n\}$ is E-satisfiable.

For example, if E is the equality theory *NT*, then $\rightsquigarrow$ may be the E-unification algorithm *UN*.

Now, we can extend SLD-resolution to take into consideration reflection principles. We use the name SLD$^{\mathcal{R}}$-resolution for the extended SLD-resolution. Recall that given a reflection principle $\mathcal{R}$, we write $\Delta_{\mathcal{R}}$ to indicate any procedure that computes $\mathcal{R}$.

▶ **SLD$^{\mathcal{R}}$-resolution**. Let $\mathcal{R}$ be a reflection principle, (P, E) a logic program and $\rightsquigarrow$ an E-unification algorithm for E. Let D be a definite clause in P and S a state $\langle M \cup \{A\}, H, F\rangle$.

Then, the state $\langle M \cup \{A_1, \ldots, A_m\}, H', F'\rangle$ is derived from S and C using $\Delta_{\mathcal{R}}$ if:

1. C is a variant of a definite clause in $\{D\} \cup \Delta_{\mathcal{R}}(D)$. Suppose that C is $p(t'_1, \ldots, t'_n) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$,
2. $(A\widehat{H})\widehat{F}$ is $p(t_1, \ldots, t_n)$, and
3. $\langle H, F, \{t_1 = t'_1, \ldots, t_n = t'_n, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H', F', \{\}\rangle$.

The atom A above is called the *selected atom* and C is called the *input clause*. The set $\{t_1 = t'_1, \ldots, t_n = t'_n, e_1, \ldots, e_q\}$ is the set of equations *added* (to H and F) by the step of SLD$^{\mathcal{R}}$-resolution.

Note that the input clause C is a variant either of a clause D in P or of a clause in $\Delta_{\mathcal{R}}(D)$. The latter case corresponds to the use of reflection axioms obtained by means of $\Delta_{\mathcal{R}}$. Observe also that we apply the substitutions $\widehat{H}$ and $\widehat{F}$ to the selected atom A. This is necessary because A may contain variables and metavariables bound in $H \cup F$. Since such variables occur both in $H \cup F$ and in $\{t_1 = t'_1, \ldots, t_n = t'_n, e_1, \ldots, e_q\}$, the E-unification algorithm would not be able to transform $\langle H, F, \{t_1 = t'_1, \ldots, t_n = t'_n, e_1, \ldots, e_q\}\rangle$ (even though it is E-satisfiable) to a tuple in quasi-solved form.

Below, we define an SLD$^{\mathcal{R}}$-derivation as a (finite or infinite) path in the tree of states, and an SLD$^{\mathcal{R}}$-refutation as a finite path in the tree ending with a success state.

▶ Let $\mathcal{R}$ be a reflection principle, (P, E) a logic program and G a definite goal $\leftarrow A_1, \ldots, A_k$. Let $\langle H_0, F_0, \{\}\rangle$ be a tuple in quasi-solved form. An **SLD$^{\mathcal{R}}$-derivation** of $(P, E) \cup \{G\}$ starting with (H_0, F_0) consists of a (finite or infinite) sequence of states $\langle\{A_1, \ldots, A_k\}, H_0, F_0\rangle$, $\langle M_1, H_1, F_1\rangle, \ldots$ and a sequence $C_1, C_2, \ldots$ of input clauses 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} using $\Delta_{\mathcal{R}}$.

▶ Let $\mathcal{R}$ be a reflection principle, (P, E) a logic program and G a definite goal. Suppose that $\langle H, F, \{\}\rangle$ is a tuple in quasi-solved form. Then, an **SLD$^{\mathcal{R}}$-refutation** of $(P, E) \cup \{G\}$ starting with (H, F) is a finite SLD$^{\mathcal{R}}$-derivation of $(P, E) \cup \{G\}$ starting with (H, F) which has a success state as the 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.

Example 5.12 Consider the reflection principle $\mathcal{R}$ and the logic program (P, E) of Example 5.3. An SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow q\}$ starting with $(\{\}, \{\})$ is the following.

- The initial state is $S_0 = \langle\{q\}, \{\}, \{\}\rangle$.
- By applying one step of SLD$^{\mathcal{R}}$-resolution, from S_0 and the first clause in P we derive the state $S_1 = \langle\{\mathit{demo}(p^1)\}, \{\}, \{\}\rangle$.
- Finally, by applying $\Delta_{\mathcal{R}}$ to the second definite clause in P, that is, $\Delta_{\mathcal{R}}(p) = \{\mathit{demo}(p^1)\}$, from S_1 and the input clause $\mathit{demo}(p^1)$ we obtain the success state $S_2 = \langle\{\}, \{\}, \{\}\rangle$.

$\square$

Example 5.13 Let (P, E) be the following logic program:

$$\left(\left\{\begin{array}{l} p(x) \leftarrow q(x) \\ q(a) \end{array}\right\}, NT\right),$$

and let $\mathcal{R}$ be:

$$\begin{array}{lcl} \mathcal{R}(p(x) \leftarrow q(x)) & = & \{\mathit{demo}([p^1, X]) \leftarrow X = \uparrow x, q(x)\} \\ \mathcal{R}(q(a)) & = & \{\mathit{demo}([p^1, X]) \leftarrow X = \uparrow a\}. \end{array}$$

Then, an SLD$^{\mathcal{R}}$-refutation for $(P, E) \cup \{\leftarrow \mathit{demo}([p^1, X])\}$ starting with $(\{\}, \{\})$ is as follows.

- The initial state is $S_0 = \langle\{\mathit{demo}([p^1, X])\}, \{\}, \{\}\rangle$.
- From S_0 and the input clause $\mathit{demo}([p^1, X]) \leftarrow X = \uparrow x, q(x)$ we derive the state $S_1 = \langle\{q(x)\}, \{\}, \{X = \uparrow x\}\rangle$.
- Finally, by considering the second clause in P, we obtain the success state $S_2 = \langle\{\}, \{x = a, X = a^1\}, \{\}\rangle$.

$\square$

Properties of SLD$^{\mathcal{R}}$-resolution

In this chapter, we present the results of soundness and completeness of SLD$^{\mathcal{R}}$-resolution with respect to the least reflective E-model. These results hold if the E-unification algorithm that is the parameter of SLD$^{\mathcal{R}}$-resolution is terminating and sound for E. In fact, not adding any E-solution (during the unification process) allow us to have soundness of SLD$^{\mathcal{R}}$-resolution, and not losing any E-solution provides completeness.

Recall that we consider equality theories that are sufficiently complete.

6.1 SOUNDNESS

To prove soundness of SLD$^{\mathcal{R}}$-resolution we use the following definition and the next lemma.

► Let (P, E) be a logic program and G a definite goal. Suppose that $\langle \{\}, H, F\rangle$ is the success state of an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$. Then, $\langle H, F\rangle$ is a **computed answer** for $(P, E) \cup \{G\}$.

Recall that we indicate a Herbrand assignment and its corresponding substitution with H and $\widehat{H}$, respectively. Thus, given a substitution $\widehat{H} = \{x_1/t_1, \ldots, x_k/t_k\}$, the corresponding Herbrand assignment H is $\{x_1 = t_1, \ldots, x_k = t_k\}$.

Lemma 6.1 *Let E be an equality theory, S_1 and S_2 two sets of equations and $\langle H_1, F_1, \{\}\rangle$ a tuple in quasi-solved form. Suppose that*

$$\langle H_1, F_1, S_1\rangle \rightsquigarrow \langle H_2, F_2, \{\}\rangle \text{ and } \langle H_2, F_2, S_2\rangle \rightsquigarrow \langle H_3, F_3, \{\}\rangle.$$

Then, $E \models \forall(S_1\widehat{H_3}\widehat{H_3'})$, for every E-solution H_3' of F_3.

Proof. Let H'_3 be any E-solution of F_3 and let $\widehat{H}$ be $\widehat{H_3}\widehat{H'_3}$, i.e., $\widehat{H}$ is the composition of the substitutions $\widehat{H_3}$ and $\widehat{H'_3}$.

Since $\langle H_1, F_1, S_1\rangle \leadsto \langle H_2, F_2, \{\}\rangle$ and $\langle H_2, F_2, S_2\rangle \leadsto \langle H_3, F_3, \{\}\rangle$, by the soundness of $\leadsto$, every E-solution of $H_2 \cup F_2$ is an E-solution of S_1 and every E-solution of $H_3 \cup F_3$ is an E-solution of $H_2 \cup F_2$. Thus, every E-solution of $H_3 \cup F_3$ is an E-solution of S_1.

To prove the statement, it suffices to show that H is an E-solution of $H_3 \cup F_3$.

Let $H_3 = \{x_1 = t_1, \dots, x_n = t_n\}$ and $H'_3 = \{y_1 = s_1, \dots, y_m = s_m\}$. By the definition of composition of substitutions, H is an Herbrand assignment containing the bindings $x_i = t_i\widehat{H'_3}$ such that $x_i \not\equiv t_i\widehat{H'_3}$ (with $1 \leq i \leq n$) and the bindings $y_j = s_j$ (with $1 \leq j \leq m$) in H'_3 such that $y_j \notin \{x_1, \dots, x_n\}$.

By noting that $\langle H_3, F_3, \{\}\rangle$ is in quasi-solved form, it follows that H is an E-solution of $H_3 \cup F_3$. ⊠

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

Theorem 6.2 (Soundness of SLD$^{\mathcal{R}}$-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 $\langle H, F, \{\}\rangle$ be any tuple in quasi-solved form and G the goal $\leftarrow A_1, \dots, A_k$. Suppose that $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F) of the form $\langle\{A_1, \dots, A_k\}, H, F\rangle, \langle M_1, H_1, F_1\rangle, \dots, \langle\{\}, H_n, F_n\rangle$.

To prove the statement, we have to show that, for every E-solution H'_n of F_n, $\forall((A_1 \wedge \dots \wedge A_k)\widehat{H_n}\widehat{H'_n})$ is a reflective logical E-consequence of (P, E). The result is proved by induction on the length n of the SLD$^{\mathcal{R}}$-refutation.

Let $\leadsto$ be any E-unification algorithm for E.

Base Case $(n = 1)$

This means that G is a goal of the form $\leftarrow p(t_1, \dots, t_h)$, the input clause is a unit clause of the form $p(t'_1, \dots, t'_h) \leftarrow e_1, \dots, e_q$ (with $q \geq 0$) and $\langle H, F, \{t_1 = t'_1, \dots, t_h = t'_h, e_1, \dots, e_q\}\rangle \leadsto \langle H_1, F_1, \{\}\rangle$.

By the soundness of $\leadsto$, $E \models \forall((t_1 = t'_1 \wedge \dots \wedge t_h = t'_h \wedge e_1 \wedge \dots \wedge e_q)\widehat{H_1}\widehat{H'_1})$, for every E-solution H'_1 of F_1. Thus, if $q > 0$, $E \models \forall((e_1 \wedge \dots \wedge e_q)\widehat{H_1}\widehat{H'_1})$ and, consequently, $\forall(p(t'_1, \dots, t'_h)\widehat{H_1}\widehat{H'_1})$ is a reflective logical E-consequence of (P, E). Hence, $\forall(p(t_1, \dots, t_h)\widehat{H_1}\widehat{H'_1})$ is a reflective logical E-consequence of

(P, E) because $\forall(p(t_1, \ldots, t_h)\widehat{H_1}\widehat{H'_1})$ is the same as $\forall(p(t'_1, \ldots, t'_h)\widehat{H_1}\widehat{H'_1})$ by $E \models \forall((t_1 = t'_1 \wedge \ldots \wedge t_h = t'_h)\widehat{H_1}\widehat{H'_1})$.

Inductive Step

Assume that the result holds for computed answers coming from SLD$^{\mathcal{R}}$-refutations of length $n-1$. Suppose that $\langle H_n, F_n\rangle$ is the computed answer of an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F) of length n. Let $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r$ be the first input clause and A_m the selected atom of G. Suppose that $(A_m\widehat{H})\widehat{F}$ is $p(t_1, \ldots, t_h)$.

This means that $\langle H, F, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$ and the tuple $\langle H_1, F_1, \{\}\rangle$ is in quasi-solved form. The derived state is $\langle\{A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}, H_1, F_1\rangle$.

Since $(P, E) \cup \{\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H_1, F_1) of length $n-1$ with computed answer $\langle H_n, F_n\rangle$, by the induction hypothesis, for every E-solution H'_n of F_n, $\forall((A_1 \wedge \ldots \wedge A_{m-1} \wedge B_1 \wedge \ldots \wedge B_r \wedge A_{m+1} \wedge \ldots \wedge A_k)\widehat{H_n}\widehat{H'_n})$ is a reflective logical E-consequence of (P, E). Consequently, if $r > 0$, $\forall((B_1 \wedge \ldots \wedge B_r)\widehat{H_n}\widehat{H'_n})$ is a reflective logical E-consequence of (P, E).

Let *SE* be the union of all the sets of equations added at every step of the SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}$ starting with (H_1, F_1). Clearly, $\langle H_1, F_1, SE\rangle \rightsquigarrow \langle H_n, F_n, \{\}\rangle$ (or, equivalently, $\langle H_1, F_1, SE\rangle$ is transformed into a tuple having the same set of E-solutions of $\langle H_n, F_n, \{\}\rangle$).

Since $\langle H, F, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$, the tuple $\langle H, F, \{\}\rangle$ is in quasi-solved form and $\langle H_1, F_1, SE\rangle \rightsquigarrow \langle H_n, F_n, \{\}\rangle$, it follows from Lemma 6.1 that $E \models \forall((t_1 = t'_1 \wedge \ldots \wedge t_h = t'_h \wedge e_1 \wedge \ldots \wedge e_q)\widehat{H_n}\widehat{H'_n})$.

Hence, $\forall(p(t_1, \ldots, t_h)\widehat{H_n}\widehat{H'_n})$ is a reflective logical E-consequence of (P, E) and, in turn, $\forall((A_1 \wedge \ldots \wedge A_k)\widehat{H_n}\widehat{H'_n})$ is a reflective logical E-consequence of (P, E). ⊠

Furthermore, the following result is an immediate consequence.

Corollary 6.3 *Let (P, E) be a logic program and G a definite goal. Suppose that there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$. Then, $(P, E) \cup \{G\}$ is reflectively E-unsatisfiable.*

Proof. Let G be a goal of the form $\leftarrow A_1, \ldots, A_k$. By Theorem 6.2, every computed answer $\langle H, F\rangle$ of $(P, E) \cup \{G\}$ is correct. Thus, for every E-

solution H' of F, $\forall((A_1 \wedge \ldots \wedge A_k)\widehat{H}\widehat{H'})$ is a reflective logical E-consequence of (P, E). Hence, $(P, E) \cup \{G\}$ is reflectively E-unsatisfiable. ⊠

▶ 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$^{\mathcal{R}}$-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 $\lceil A \rceil$.)

The converse of the next corollary also holds (see Theorem 6.9).

Corollary 6.4 *If an atom A belongs to the success set of a logic program (P, E), then $\lceil A \rceil$ is contained in the least reflective E-model of (P, E).*

Proof. Suppose that $(P, E) \cup \{\leftarrow A\}$ has an SLD$^{\mathcal{R}}$-refutation with computed answer $\langle H, F \rangle$. Since, by the definition of success set, A is ground, by Theorem 6.2 A is a reflective logical E-consequence of (P, E). Hence, $\lceil A \rceil$ is in the least reflective E-model of (P, E). ⊠

Now we strengthen Corollary 6.4 by showing that, if a ground atom A has an SLD$^{\mathcal{R}}$-refutation of length n, then $\lceil A \rceil \in T^{\mathcal{R}}_{(P,E)}{\uparrow}n$. This is an extension of the result due to Apt & van Emden [7].

We use the following definition.

▶ The **closure** of an atom A, written as $\Psi(A)$, is the set of representative elements of all ground instances of A,

$$\Psi(A) = \{\lceil B \rceil \mid \text{ for every ground instance } B \text{ of } A\}.$$

Theorem 6.5 *Let (P, E) be a logic program and G a definite goal of the form $\leftarrow A_1, \ldots, A_k$. Suppose $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation of length n with computed answer $\langle H_n, F_n \rangle$. Then, $\bigcup_{j=1}^{k} \Psi(A_j \widehat{H_n} \widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}{\uparrow}n$, for every E-solution H'_n of F_n.*

Proof. Let $\langle H, F, \{\}\rangle$ be any tuple in quasi-solved form. Suppose that $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F) of the form $\langle\{A_1, \ldots, A_k\}, H, F\rangle, \langle M_1, H_1, F_1\rangle, \ldots, \langle\{\}, H_n, F_n\rangle$.

The result is proved by induction on the length n of the SLD$^{\mathcal{R}}$-refutation. Let $\leadsto$ be any E-unification algorithm for E.

Base Case $(n = 1)$

This means that G is a goal $\leftarrow p(t_1, \ldots, t_h)$, the input clause is a unit clause $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q$ and $\langle H, F, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \leadsto \langle H_1, F_1, \{\}\rangle$.

By the soundness of $\leadsto$, $E \models \forall((t_1 = t'_1 \wedge \ldots \wedge t_h = t'_h \wedge e_1 \wedge \ldots \wedge e_q)\widehat{H_1}\widehat{H'_1})$, for every E-solution H'_1 of F_1. Clearly, $\Psi(p(t'_1, \ldots, t'_h)\widehat{H_1}\widehat{H'_1}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow 1$ by the definition of $T^{\mathcal{R}}_{(P,E)}$, and so $\Psi(p(t_1, \ldots, t_h)\widehat{H_1}\widehat{H'_1}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow 1$.

Inductive step

Suppose that the result holds for SLD$^{\mathcal{R}}$-refutations of length $n-1$. Consider an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F) of length n. Let A_j be an atom of G. We distinguish between two cases depending on whether or not A_j is the selected atom of G.

Case 1 (A_j is not the selected atom in G)

Then, A_j is an atom of M_1, where $\langle M_1, H_1, F_1\rangle$ is the second state of the SLD$^{\mathcal{R}}$-refutation. Suppose $M_1 = \{D_1, \ldots, D_l\}$.

Since $(P, E) \cup \{\leftarrow D_1, \ldots, D_l\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H_1, F_1) of length $n - 1$ with computed answer $\langle H_n, F_n\rangle$, by the induction hypothesis $\Psi(A_j\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow(n-1)$, for every E-solution H'_n of F_n. By the monotonicity of $T^{\mathcal{R}}_{(P,E)}$, we have that $\Psi(A_j\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow n$.

Case 2 (A_j is the selected atom in G)

Let $A_j\widehat{H}\widehat{F}$ be the atom $p(t_1, \ldots, t_h)$. Suppose that the first input clause is $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r$ $(q \geq 0, r \geq 0)$. This means that $\langle H, F, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \leadsto \langle H_1, F_1, \{\}\rangle$.

Let SE be the union of all the sets of equations added at every step of the SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow A_1, \ldots, A_{j-1}, B_1, \ldots, B_r, A_{j+1}, \ldots, A_k\}$ starting with (H_1, F_1). Clearly, $\langle H_1, F_1, SE\rangle \leadsto \langle H_n, F_n, \{\}\rangle$ (or, equivalently, $\langle H_1, F_1, SE\rangle$ is transformed into a tuple having the same set of E-solutions of $\langle H_n, F_n, \{\}\rangle$).

Since $\langle H, F, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$, the tuple $\langle H, F, \{\}\rangle$ is in quasi-solved form and $\langle H_1, F_1, SE\rangle \rightsquigarrow \langle H_n, F_n, \{\}\rangle$, it follows from Lemma 6.1 that $E \models \forall((t_1 = t'_1 \wedge \ldots \wedge t_h = t'_h \wedge e_1 \wedge \ldots \wedge e_q)\widehat{H_n}\widehat{H'_n})$, for every E-solution H'_n of F_n.

If $r = 0$, $\Psi(p(t'_1, \ldots, t'_h)\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow 1$. Thus, $\Psi(p(t_1, \ldots, t_h)\widehat{H_n}\widehat{H'_n}) = \Psi(p(t'_1, \ldots, t'_h)\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow 1 \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow n$.

If $r > 0$, by the induction hypothesis, $\Psi(B_i\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow(n-1)$, for all $1 \leq i \leq r$. Since $E \models \forall((t_1 = t'_1 \wedge \ldots \wedge t_h = t'_h \wedge e_1 \wedge \ldots \wedge e_q)\widehat{H_n}\widehat{H'_n})$, we have that $\Psi(p(t_1, \ldots, t_h)\widehat{H_n}\widehat{H'_n}) \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow n$. ⊠

6.2 COMPLETENESS

The main result of this section is the completeness of SLD$^{\mathcal{R}}$-resolution. We begin our argument for completeness by appropriately rephrasing the Lifting lemma [79].

We use the following definition and the next lemma.

▶ Let E be an equality theory, A and B two sets of equations. A and B are **equivalent**, written as $A \approx B$, if they admit the same set of E-solutions.

Example 6.6 Consider the equality theory NT. Given the sets:

$$\begin{array}{ll} A = \{X = \uparrow y\} & B = \{x = f(Y)\} \\ A' = \{\downarrow X = y\} & B' = \{x = f(a^1), Y = a^1\}, \end{array}$$

we have that $A \approx A'$ and $B \not\approx B'$. B is "more general" than B' since B admits more E-solutions than B'. In fact, every E-solution of B' is also an E-solution of B, but not vice versa. We can "restrict" B with a set S of equations to make them equivalent. Let $S = \{Y = a^1\}$, then $B \cup S \approx B'$.

□

Lemma 6.7 *Let (P, E) be a logic program and G a definite goal. Let $\langle H, F, \{\}\rangle$ and $\langle H', F', \{\}\rangle$ be tuples in quasi-solved form. Let S be a set of equations such that*

$$H \cup F \approx S \cup H' \cup F'.$$

Suppose that $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F) with computed answer $\langle H_n, F_n\rangle$. Then, there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H', F') of the same length with computed answer $\langle H'_n, F'_n\rangle$ such that

$$H_n \cup F_n \approx S \cup H'_n \cup F'_n.$$

The proof of the lemma uses the following two properties. Let A, B and C be any sets of equations and $\langle H, F, \{\}\rangle$ a tuple in quasi-solved form.

$$H \cup F \cup A \approx H \cup F \cup A\widehat{H}\widehat{F} \qquad (\dagger)$$
$$A \approx B \text{ implies } A \cup C \approx B \cup C \qquad (\ddagger)$$

Proof. Let G be $\leftarrow A_1, \ldots, A_k$. Suppose that $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F) of the form

$$\langle\{A_1, \ldots, A_k\}, H, F\rangle, \langle M_1, H_1, F_1\rangle, \ldots, \langle\{\}, H_n, F_n\rangle$$

of length n with input clauses $C_1, \ldots, C_n$. We may assume that $\widehat{H}$, $\widehat{F}$, $\widehat{H'}$ and $\widehat{F'}$ do not act on the variables and metavariables of any input clause.

The proof is by induction on the length n of the SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F).

Let $\rightsquigarrow$ be any E-unification algorithm for E.

Base Case ($n = 1$)

This means that G is of the form $\leftarrow p(t_1, \ldots, t_h)$, the atom $p(t_1, \ldots, t_h)\widehat{H}\widehat{F}$ is $p(s_1, \ldots, s_h)$, the input clause C_1 is a unit clause $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q$ and $\langle H, F, \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$.
Clearly, $H \cup F \cup \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable.

Let $p(t_1, \ldots, t_h)\widehat{H'}\widehat{F'}$ be $p(u_1, \ldots, u_h)$. Then, it holds that:

$$\begin{array}{lr}
H \cup F \cup \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\} & \\
\qquad\qquad \equiv & (*) \\
H \cup F \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\widehat{H}\widehat{F} & \\
\qquad\qquad \approx & (\text{by } \dagger) \\
H \cup F \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\} & \\
\qquad\qquad \approx & (\text{by } \ddagger) \\
S \cup H' \cup F' \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\} & \\
\qquad\qquad \approx & (\text{by } \dagger \text{ and } \ddagger) \\
S \cup H' \cup F' \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\widehat{H'}\widehat{F'} & \\
\qquad\qquad \equiv & (**) \\
S \cup H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}. &
\end{array}$$

The steps $(*)$ and $(**)$ hold by the assumption that $\widehat{H}$, $\widehat{F}$, $\widehat{H'}$ and $\widehat{F'}$ do not act on the variables and metavariables of C_1.

Since $H \cup F \cup \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable and

$$\begin{array}{c}
H \cup F \cup \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\} \\
\approx \\
S \cup H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\},
\end{array}$$

the set $S \cup H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable. Consequently, $H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable and, by the definition of $\rightsquigarrow$,

$$\langle H', F', \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H'_1, F'_1, \{\}\rangle,$$

for some tuple $\langle H'_1, F'_1, \{\}\rangle$ in quasi-solved form.

Hence, $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H', F') with input clause C_1. It holds that:

$$\begin{aligned} H_1 \cup F_1 &\approx H \cup F \cup \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\} \\ &\approx S \cup H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\} \\ &\approx S \cup H'_1 \cup F'_1. \end{aligned}$$

Inductive Step

Assume that the result holds for SLD$^{\mathcal{R}}$-refutations of length $n-1$. Consider an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F) of length n with computed answer $\langle H_n, F_n\rangle$ and with input clauses $C_1, \ldots, C_n$. Let C_1 be $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r$ and A_m the selected atom of G. Suppose that A_m is $p(t_1, \ldots, t_h)$ and $A_m\widehat{H}\widehat{F}$ is $p(s_1, \ldots, s_h)$.

This means that $\langle H, F, \{s_1 = t'_1, \ldots, s_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$ and the tuple $\langle H_1, F_1, \{\}\rangle$ is in quasi-solved form. The derived state is $\langle\{A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}, H_1, F_1\rangle$.

Let $A_m\widehat{H'}\widehat{F'}$ be $p(u_1, \ldots, u_h)$. Analogously to the argumentation in the base case, $H' \cup F' \cup \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable and, consequently, $\langle H', F', \{u_1 = t'_1, \ldots, u_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H'_1, F'_1, \{\}\rangle$, for some tuple $\langle H'_1, F'_1, \{\}\rangle$ in quasi-solved form. Trivially,

$$H_1 \cup F_1 \approx S \cup H'_1 \cup F'_1.$$

Thus, the state $\langle\{A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}, H'_1, F'_1\rangle$ can be derived from $\langle\{A_1, \ldots, A_k\}, H', F'\rangle$, input clause C_1 and selected atom A_m.

By noting that both $\langle H_1, F_1, \{\}\rangle$ and $\langle H'_1, F'_1, \{\}\rangle$ are in quasi-solved form, $H_1 \cup F_1 \approx S \cup H'_1 \cup F'_1$ and

$$(P, E) \cup \{\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}$$

has an SLD$^{\mathcal{R}}$-refutation starting with (H_1, F_1) of length $n-1$, by the induction hypothesis $(P, E) \cup \{\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, \ldots, A_k\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H'_1, F'_1) of length $n-1$ with computed answer $\langle H'_n, F'_n\rangle$ such that

$$H_n \cup F_n \approx S \cup H'_n \cup F'_n.$$

The claim follows. ⊠

Now, we can state our main lemma. This lemma essentially states that, if we can prove an instance $G\widehat{H}\widehat{F}$ of a goal G from a logic program (P, E) (i.e., there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F)), then we can also prove the goal G (i.e., there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with $(\{\}, \{\})$). The two proofs have the same length and are such that the computed answer of $G\widehat{H}\widehat{F}$ is equivalent to the computed answer of G by taking into consideration the bindings contained in $H \cup F$.

Lemma 6.8 (Lifting) *Let (P, E) be a logic program and G a definite goal. Let $\langle H, F, \{\}\rangle$ be a tuple in quasi-solved form. Suppose there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with (H, F) with computed answer $\langle H_n, F_n\rangle$. Then, there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{G\}$ starting with $(\{\}, \{\})$ of the same length with computed answer $\langle H'_n, F'_n\rangle$ such that*

$$H_n \cup F_n \approx H \cup F \cup H'_n \cup F'_n.$$

Proof. Immediate by Lemma 6.7. ⊠

The first completeness result gives the converse of Corollary 6.4.

Theorem 6.9 *Let (P, E) be a logic program. An atom A belongs to the success set of (P, E) if and only if $\lceil A\rceil$ is contained in the least reflective E-model of (P, E).*

Proof. By Corollary 6.4, it suffices to show that, if $\lceil A\rceil$ belongs to the least reflective E-model of (P, E), then A is contained in the success set of (P, E). Suppose that $\lceil A\rceil$ is in the least reflective E-model of (P, E). Then by Theorem 5.8, $\lceil A\rceil \in T^{\mathcal{R}}_{(P,E)}\uparrow n$, for some $n \in \omega$. We prove by induction on n that $\lceil A\rceil \in T^{\mathcal{R}}_{(P,E)}\uparrow n$ implies that $(P, E) \cup \{\leftarrow A\}$ has a SLD$^{\mathcal{R}}$-refutation and hence A is in the success set of (P, E).

Note that A is ground by the definition of success set. Let A be $p(t_1, \ldots, t_h)$. Let $\rightsquigarrow$ be any E-unification algorithm for E.

Base Case $(n = 1)$

Then, $\lceil A\rceil \in T^{\mathcal{R}}_{(P,E)}\uparrow 1$.

By the definition of $T^{\mathcal{R}}_{(P,E)}$, there exists a ground instance $(p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q)\widehat{H}\widehat{F}$ of a unit clause $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q$ in $P \cup \mathcal{R}(P)$ such that $\lceil p(t_1, \ldots, t_h)\rceil \equiv \lceil p(t'_1, \ldots, t'_h)\widehat{H}\widehat{F}\rceil$ and $E \models (e_1 \wedge \ldots \wedge e_q)\widehat{H}\widehat{F}$, for some tuple $\langle H, F, \{\}\rangle$ in quasi-solved form.

Clearly, $\{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable and consequently, by the soundness of $\rightsquigarrow$, we have that

$$\langle \{\}, \{\}, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle,$$

for some tuple $\langle H_1, F_1, \{\}\rangle$ in quasi-solved form. Hence, $(P, E) \cup \{\leftarrow A\}$ has an SLD$^{\mathcal{R}}$-refutation (starting with $(\{\}, \{\})$).

Inductive Step

Suppose that the result holds for $n-1$. Assume that $\lceil A \rceil \in T^{\mathcal{R}}_{(P,E)}\uparrow n$.

By the definition of $T^{\mathcal{R}}_{(P,E)}$, there exists a ground instance of the form $(p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_m)\widehat{H}\widehat{F}$ of a clause in $P \cup \mathcal{R}(P)$ such that $\lceil p(t_1, \ldots, t_h)\rceil \equiv \lceil p(t'_1, \ldots, t'_h)\widehat{H}\widehat{F}\rceil$, $E \models (e_1 \wedge \ldots \wedge e_q)\widehat{H}\widehat{F}$ and

$$\{\lceil B_1\widehat{H}\widehat{F}\rceil, \ldots, \lceil B_m\widehat{H}\widehat{F}\rceil\} \subseteq T^{\mathcal{R}}_{(P,E)}\uparrow(n-1),$$

for some tuple $\langle H, F, \{\}\rangle$ in quasi-solved form.

Clearly, the set $\{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}$ is E-satisfiable because $\lceil p(t_1, \ldots, t_h)\rceil \equiv \lceil p(t'_1, \ldots, t'_h)\widehat{H}\widehat{F}\rceil$ and $E \models (e_1 \wedge \ldots \wedge e_q)\widehat{H}\widehat{F}$. Consequently, $\langle \{\}, \{\}, \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\rangle \rightsquigarrow \langle H_1, F_1, \{\}\rangle$, for some tuple $\langle H_1, F_1, \{\}\rangle$ in quasi-solved form.

By the induction hypothesis, $(P, E) \cup \{\leftarrow B_i\widehat{H}\widehat{F}\}$ has an SLD$^{\mathcal{R}}$-refutation with computed answer $\langle H_i, F_i\rangle$, for $1 \leq i \leq m$. As each $B_i\widehat{H}\widehat{F}$ is ground, every computed answer $\langle H_i, F_i\rangle$ contains variables and metavariables that are distinct from those of the computed answers of the other SLD$^{\mathcal{R}}$-refutations. Thus, these SLD$^{\mathcal{R}}$-refutations can be combined into an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow (B_1, \ldots, B_m)\widehat{H}\widehat{F}\}$. Equivalently, by the definition of SLD$^{\mathcal{R}}$-resolution, $(P, E) \cup \{\leftarrow B_1, \ldots, B_m\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F).

By noting that the set $\{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\widehat{H}\widehat{F}$ is E-satisfiable and contains ground equations, every Herbrand assignment is an E-solution of the set. Consequently, it holds that:

$$\begin{array}{rcl} H \cup F & \approx & H \cup F \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\}\widehat{H}\widehat{F} \\ & \approx & H \cup F \cup \{t_1 = t'_1, \ldots, t_h = t'_h, e_1, \ldots, e_q\} \\ & \approx & H \cup F \cup H_1 \cup F_1. \end{array}$$

Now, by applying one step of SLD$^{\mathcal{R}}$-resolution, from the state $\langle \{A\}, \{\}, \{\}\rangle$ and the definite clause $p(t'_1, \ldots, t'_h) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_m$ we derive the state $\langle \{B_1, \ldots, B_m\}, H_1, F_1\rangle$.

Since we have that $\langle H, F, \{\}\rangle$ and $\langle H_1, F_1, \{\}\rangle$ are in quasi-solved form, $H \cup F \approx H \cup F \cup H_1 \cup F_1$ and $(P, E) \cup \{\leftarrow B_1, \ldots, B_m\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F), it follows by Lemma 6.7 that $(P, E) \cup \{\leftarrow B_1, \ldots, B_m\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H_1, F_1). Hence, $(P, E) \cup \{\leftarrow A\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(\{\}, \{\})$. ⊠

Theorem 6.10 *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$^{\mathcal{R}}$-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, $\forall\neg(\lceil A_1 \rceil \wedge \ldots \wedge \lceil A_k \rceil)$ is false with respect to the least reflective E-model $M^{\mathcal{R}}_{(P,E)}$. Hence, there exists some ground instance $G\widehat{H}\widehat{F}$ of G such that $\neg(\lceil A_1\widehat{H}\widehat{F} \rceil \wedge \ldots \wedge \lceil A_k\widehat{H}\widehat{F} \rceil)$ is false with respect to $M^{\mathcal{R}}_{(P,E)}$, for some tuple $\langle H, F, \{\}\rangle$ in quasi-solved form. Consequently, we have that $\{\lceil A_1\widehat{H}\widehat{F} \rceil, \ldots, \lceil A_k\widehat{H}\widehat{F} \rceil\} \subseteq M^{\mathcal{R}}_{(P,E)}$. By Theorem 6.9, there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow A_i\widehat{H}\widehat{F}\}$ with computed answer $\langle H_i, F_i\rangle$, for every $1 \leq i \leq k$. As each $A_i\widehat{H}\widehat{F}$ is ground, every computed answer $\langle H_i, F_i\rangle$ contains variables and metavariables that are distinct from those of the other computed answers. Thus, these SLD$^{\mathcal{R}}$-refutations can be combined into an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow (A_1, \ldots, A_k)\widehat{H}\widehat{F}\}$. Equivalently, by the definition of SLD$^{\mathcal{R}}$-resolution, $(P, E) \cup \{\leftarrow A_1, \ldots, A_k\}$ has an SLD$^{\mathcal{R}}$-refutation starting with (H, F). Finally, we apply Lemma 6.8. ⊠

Next we turn attention to correct answers. It is not possible to prove the exact converse of Theorem 6.2 because computed answers are always more "general" than correct answers with respect to the variables and 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.

Let $x_1, \ldots, x_n$ be all the variables and metavariables in F. We abbreviate $\forall x_1 \ldots \forall x_m \exists x_{m+1} \ldots \exists x_n F$ as $\forall x_1 \ldots \forall x_m \exists F$, i.e., every $x_i \notin \{x_1, \ldots, x_m\}$ is existentially quantified.

Lemma 6.11 *Let (P, E) be a logic program and A an atom. Suppose that $x_1, \ldots, x_n$ are all the variables and 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$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\leftarrow A\}$ starting with $(\{\}, \{\})$ with computed answer $\langle H'_n, F'_n\rangle$ such that the atom $A\widehat{H'_n}$ is a variant of A and $E \models \forall x_1 \ldots \forall x_n \exists F'_n$.*

Proof. Let $a_1, \ldots, a_n$ be distinct constants and metaconstants of the language occurring neither in P nor in A. Let H' be the Herbrand assignment

$\{x_1 = a_1, \ldots, x_n = a_n\}$. Then, $A\widehat{H'}$ is a reflective logical E-consequence of (P, E).

As $A\widehat{H'}$ is ground, Theorem 6.9 states that $(P, E) \cup \{\leftarrow A\widehat{H'}\}$ has an SLD$^{\mathcal{R}}$-refutation. By the definition of SLD$^{\mathcal{R}}$-resolution, $(P, E) \cup \{\leftarrow A\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(H', \{\})$ with computed answer $\langle H_n, F_n \rangle$. By Lemma 6.8, $(P, E) \cup \{\leftarrow A\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(\{\}, \{\})$ with computed answer $\langle H'_n, F'_n \rangle$ such that $H_n \cup F_n \approx H' \cup H'_n \cup F'_n$. By the definition of computed answer, $H_n \cup F_n$ is E-satisfiable and so $H' \cup H'_n \cup F'_n$ is E-satisfiable.

Finally, by noting that every x_i (with $1 \leq i \leq n$) in H' is bound to a constant or a metaconstant a_i that occurs neither in P nor in A, and that $H' \cup H'_n \cup F'_n$ is E-satisfiable, it follows that, if H'_n contains a binding for x_i, then such a binding has the form $x_i = y_j$, for some variable or metavariable y_j. Consequently, $A\widehat{H'_n}$ is a variant of A.

Furthermore, the equations in F'_n are always E-satisfiable independently from the values of $x_1, \ldots, x_n$ because $H' \cup H'_n \cup F'_n$ is E-satisfiable. Hence, it holds that $E \models \forall x_1 \ldots \forall x_n \exists F$. ⊠

Example 6.12 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 E-satisfiable for every value of x, that is, $NT \models \forall x\, \exists Y\ (Y = \uparrow x)$. A computed answer for the goal $\leftarrow p(z)$ is $\langle \{z = x\}, \{Y = \uparrow x\} \rangle$. It holds that $p(z)\{z/x\}$ is a variant of $p(z)$. □

Now, we are in the position to state the main completeness result. With this aim, we introduce the following definitions.

- ▶ Let H be a Herbrand assignment and V a set of variables and metavariables. The **restriction of** H induced by V, written as H_V, is the Herbrand assignment containing all bindings $x = t$ of H such that $x \in V$.

- ▶ Let E be an equality theory, A and B two sets of equations and V a set of variables and metavariables. A and B are **equivalent with respect to** V, written as $A \approx_V B$, if they admit the same set of E-solutions restricted by V.

The previous definition says that A and B are equivalent with respect to V if, for every E-solution H of A, there exists an E-solution H' of B such that H'_V is H_V, and the converse also holds.

Example 6.13 Consider the syntactic equality, i.e., $E = \{\}$. Given the sets:

$$\begin{array}{lll} A = \{f(x) = f(y), y = g(z)\} & B = \{x = c, y = x\} & C = \{x = y\} \\ A' = \{x = g(z), y = g(z)\} & B' = \{x = z, y = a, z = c\} & C' = \{x = z\}, \end{array}$$

we have that $A \approx_{\{x,y,z\}} A'$, $B \approx_{\{x\}} B'$, $B \not\approx_{\{x,y\}} B'$, $C \approx_{\{x\}} C'$, $C \approx_{\{y\}} C'$ and $C \not\approx_{\{x,y,z\}} C'$. □

Below, we abbreviate $vars(G)$ as $v(G)$. Recall that we write $vars(G)$ to indicate the set of variables and metavariables in G. Given the sets of variables and metavariables $V = \{x_1, \ldots, x_n\}$ and $v(G) = V \cup \{y_1, \ldots, y_m\}$, we write $\forall_V G$ and $\forall_V \exists G$ to indicate $\forall x_1 \ldots \forall x_n G$ and $\forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m G$, respectively.

Theorem 6.14 (Completeness of SLD$^{\mathcal{R}}$-resolution) *Let (P, E) be a logic program and G a definite goal. For every correct answer $\langle H, F\rangle$ for $(P, E) \cup \{G\}$, there exists (a) a computed answer $\langle H', F'\rangle$ for $(P, E) \cup \{G\}$ and (b) a set S of equations such that $H \cup F \approx_{v(G)} H' \cup F' \cup S$.*

Proof.

Proof of (a).

Let G be $\leftarrow A_1, \ldots, A_k$. Let A be any E-solution of $H \cup F$ and $A_{v(G)}$ the restriction of A induced by $v(G)$. As $\langle H, F\rangle$ is a correct answer for $(P, E) \cup \{G\}$, it follows that $\forall(\widehat{GA_{v(G)}})$ is a reflective logical E-consequence of (P, E).

By Lemma 6.11, $(P, E) \cup \{\leftarrow \widehat{A_i A_{v(G)}}\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(\{\}, \{\})$ with computed answer $\langle H_i, F_i\rangle$ such that $(\widehat{A_i A_{v(G)}})\widehat{H_i}$ is a variant of $\widehat{A_i A_{v(G)}}$ and $E \models \forall_{v(\widehat{A_i A_{v(G)}})} \exists F_i$, for every $1 \leq i \leq k$.

Since every H_i does not instantiate $\widehat{A_i A_{v(G)}}$ and $E \models \forall_{v(\widehat{A_i A_{v(G)}})} \exists F_i$, there exists an SLD$^{\mathcal{R}}$-refutation of $(P, E) \cup \{\widehat{GA_{v(G)}}\}$ starting with $(\{\}, \{\})$ with computed answer $\langle H'', F''\rangle$ such that $(\widehat{GA_{v(G)}})\widehat{H''}$ is a variant of $\widehat{GA_{v(G)}}$ and $E \models \forall_{v(\widehat{GA_{v(G)}})} \exists F''$.

Hence, by the definition of SLD$^{\mathcal{R}}$-resolution, $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(A_{v(G)}, \{\})$ with computed answer $\langle H''', F''\rangle$, where

H''' is $A_{v(G)}\widehat{H''} \cup H''$. Thus, the goal $G\widehat{H'''}$ is a variant of $G\widehat{A_{v(G)}}$ and $E \models \forall_{v(G\widehat{A_{v(G)}})}\exists F''$. This means that the variables and the metavariables occurring in $G\widehat{A_{v(G)}}$ are not instantiated by $\langle H''', F''\rangle$. (Note that F'' is retained since the SLD$^{\mathcal{R}}$-refutation starts with $(A_{v(G)}, \{\})$, i.e., with no name equations.)

By Lemma 6.8, $(P, E) \cup \{G\}$ has an SLD$^{\mathcal{R}}$-refutation starting with $(\{\}, \{\})$ with computed answer $\langle H', F'\rangle$ such that $H''' \cup F'' \approx A_{v(G)} \cup H' \cup F'$. Since the variables and the metavariables of $G\widehat{A_{v(G)}}$ are not instantiated by $\langle H''', F''\rangle$ and $H''' \cup F'' \approx A_{v(G)} \cup H' \cup F'$, it follows that those variables are not instantiated by $\langle H', F'\rangle$.

The set $H''' \cup F''$ is E-satisfiable by the definition of computed answer and so $A_{v(G)} \cup H' \cup F'$ is E-satisfiable. Since, by the definition of SLD$^{\mathcal{R}}$-resolution, the input clauses are standardized apart, we may assume that the variables and metavariables in $H' \cup F'$, except those occurring in G, are different from those occurring in $H \cup F$, that is, $v(H' \cup F') \cap v(H \cup F) \subseteq v(G)$.

Since $A_{v(G)} \cup H' \cup F'$ is E-satisfiable for every E-solution A of $H \cup F$, the bindings in $H' \cup F'$ do not further instantiate the variables and metavariables occurring in $A_{v(G)}$. (Otherwise, there exists an E-solution A' of $H \cup F$ that instantiates those variables and metavariables in a way different from $H' \cup F'$, such that $A'_{v(G)} \cup H' \cup F'$ is not E-satisfiable.) Thus, given an E-solution A of $H \cup F$, there exists an E-solution B of $(H' \cup F')\widehat{A_{v(G)}}$ such that $B_{v(A_{v(G)})} = \{\}$ (i.e., B does not contain any bindings for the variables in $A_{v(G)}$. Thus, B is the most general E-solution with respect to the variables of $A_{v(G)}$). Since the variables of $G\widehat{A_{v(G)}}$ are not instantiated by $\langle H', F'\rangle$, we have that $B_{v(G)} = \{\}$.

Proof of (b).

We have to show that there exists a set S such that $H \cup F \approx_{v(G)} H' \cup F' \cup S$. Let S be $H \cup F$. Observe that $H' \cup F' \cup H \cup F$ is E-satisfiable because the set $A_{v(G)} \cup H' \cup F'$ is E-satisfiable, for every E-solution A of $H \cup F$.

To prove that $H \cup F \approx_{v(G)} H' \cup F' \cup H \cup F$, we show that:

$(b1)$ for every E-solution C of $H' \cup F' \cup H \cup F$, there exists an E-solution A of $H \cup F$ such that $A_{v(G)}$ is $C_{v(G)}$;

$(b2)$ for every E-solution A of $H \cup F$, there exists an E-solution C of $H' \cup F' \cup H \cup F$ such that $C_{v(G)}$ is $A_{v(G)}$.

$(b1)$ Immediate by noting that every E-solution of $H' \cup F' \cup H \cup F$ is also an E-solution of $H \cup F$ (i.e., A is C itself).

($b2$) Assume that A is an E-solution of $H \cup F$. Then, $A_{v(G)} \cup H' \cup F'$ is E-satisfiable and $A_{v(G)} \cup H' \cup F' \approx A_{v(G)} \cup (H' \cup F')\widehat{A_{v(G)}}$. The set $(H' \cup F')\widehat{A}$ is identical to $(H' \cup F')\widehat{A_{v(G)}}$ because the variables and metavariables that occur both in $H' \cup F'$ and A are in $v(G)$.

Let B be an E-solution of $(H' \cup F')\widehat{A_{v(G)}}$ such that $B_{v(A_{v(G)})} = \{\}$ and $B_{v(G)} = \{\}$. Let C be $A \cup B$. Since A and B are Herbrand assignments, $(H' \cup F')\widehat{A_{v(G)}}$ is identical to $(H' \cup F')\widehat{A}$, B is an E-solution of $(H' \cup F')\widehat{A_{v(G)}}$ and $B_{v(A_{v(G)})} = \{\}$, it follows that C is a Herbrand assignment.

Clearly, C is an E-solution of $H \cup F$. Since C is a Herbrand assignment and B is an E-solution of $(H' \cup F')\widehat{A_{v(G)}}$, it follows that $A \cup B$ is an E-solution of $H' \cup F'$. Hence, C is an E-solution of $H' \cup F' \cup H \cup F$.

It follows from $B_{v(G)} = \{\}$ that $C_{v(G)}$ is $A_{v(G)}$. ⊠

The main proposal of the thesis 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 thesis are quite intricate, they serve as the behind-the-scenes sound definition and operation of the proposed system. Users of *RCL* shall 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 the next three chapters we offer examples of how to use the system capabilities in three different representation problems. Overall, we hope to show how one concept and tool (i.e., a reflection principle) 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.

Chapter 7

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 [38]. Reflective Prolog has been defined and implemented: it has been the seminal work which stimulated the first intuition of the concepts that, with time and thought, have led to the formalisation 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.

7.1 THE LANGUAGE REFLECTIVE PROLOG

Reflective Prolog [36, 38] is a metalogic programming language that extends the language of Horn clauses [70, 79] to contain higher-order-like features. Reflective Prolog has three basic features: a naming mechanism, metaevaluation clauses and a form of reflection. The naming mechanism was originally defined in a rather ad hoc fashion. Its axiomatization as an equality theory (computationally characterized by a rewrite system), which is the first step for defining Reflective Prolog in *RCL*, is described in [14]. (Reflective Prolog has compositional names defined as an equality theory along the lines of *NT*. Such equality theory is sufficiently complete.)

Metaevaluation clauses are clauses defining the predicate symbol *solve*; they allow to declaratively extend the meaning of other predicates, called *base level* predicates. In fact, since *solve* takes as argument the name of an atom, metaevaluation clauses make it possible to express sentences about that atom.

Example 7.1 The following is an example of a Reflective Prolog program.

$friend(Giorgio, Mary)$	*Base level*
$amico(Lucy, Albert)$	
$happy(x) \leftarrow friend(x, Lucy)$	
$symmetric(friend^1)$	
$symmetric(equivalent^1)$	
$equivalent(amico^1, friend^1)$	
	Metaevaluation
$solve([X, Y, Z]) \leftarrow symmetric(X), solve([X, Z, Y])$	*level*
$solve([X_1, Y, Z]) \leftarrow equivalent(X_1, X_2), solve([X_2, Y, Z])$	

The first three clauses of the base level define the relations *friend*, *amico* and *happy*. The two unit clauses following them state that the relations *friend* and *equivalent* are both symmetric. The last unit clause of the base level states that the relations *amico* and *friend* are equivalent.

The metaevaluation level consists of two solve clauses. The first clause declaratively defines the concept of symmetry in the theory: the objects whose names are Y and Z are in the relation whose name is X, provided that the relation denoted by X is asserted to be symmetric and that the object denoted by Z and Y are in the relation denoted by X. The second clause states that equivalent relations have the same extensions. □

Reflective Prolog embeds a form of logical reflection which makes these extensions effective both semantically and procedurally, by allowing deduction to be interleaved between the base level and the metaevaluation level of the theory.

Computationally, reflection in Reflective Prolog consists of the ability to dynamically change the level of computation. This is achieved by means of metaevaluation clauses that resolve base level goals, and vice versa, via an extended resolution principle called RSLD-resolution.

▶ **RSLD-resolution**. Let G be a goal $\leftarrow A_1, \ldots, A_k$ and A_m the selected atom in G. Then $\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_r, A_{m+1}, A_k$ is derived from G and C using substitution θ if and only if one of the following conditions holds:

1. C is $A \leftarrow B_1, \ldots, B_r$
   θ is an mgu of A_m and A
2. A_m is $solve(\_)$
   C is $A \leftarrow B_1, \ldots, B_r$
   A is distinct from $solve(\_)$
   θ is an mgu of A_m and $solve(A^1)$

3. A_m is distinct from $solve(\_)$
   C is $solve(t) \leftarrow B_1, \ldots, B_r$
   θ is an mgu of $solve(\overset{1}{A_m})$ and $solve(t)$

The first case corresponds to the standard SLD-resolution. The second case is a step of *object-to-meta reflection*: a base level clause is used in the course of proving an atom of the metaevaluation level. Finally, the third case is a step of *meta-to-object reflection*: a clause of the metaevaluation level is used to prove a base level atom.

The following two examples show the expressive and reasoning power of Reflective Prolog in the context of flexible query-answering systems. Several examples can be found in [15].

Example 7.2 Retrieval of properties of individuals.

Assume that a user, say Juliette, wants to retrieve some information about an individual, say Andrew, without knowing exactly what, or without knowing which kind of information is available in the database. Then Juliette would like to be able to ask vague queries of the form: "return all the properties of Andrew", "return only the properties of Andrew that satisfy these requirements", and so on.

In Reflective Prolog this kind of flexibility can be obtained by posing queries directly at the metaevaluation level. Suppose that the database contains the facts:

$$\begin{array}{l} young\_man(Andrew) \\ student(Andrew) \\ handsome(Andrew) \\ rich(Andrew) \\ \ldots \\ worker(Bob) \\ poor(Bob) \\ \ldots \end{array}$$

and assume that Juliette is interested in having all the available information about Andrew, then Juliette may ask the database:

$$\begin{array}{ll} ?\text{-} & solve([X, Andrew^1]) \\ & X = young\_man^1; \\ & X = student^1; \\ & \ldots \end{array}$$

Suppose now that Juliette wants to specialise the search in the database with respect to the properties she finds interesting in a man, for example, the following:

$$\begin{array}{l} interesting(Juliette, man, handsome^1) \\ interesting(Juliette, man, rich^1). \end{array}$$

Then, she may decide to go on a date with him after having asked the database:

$$\begin{array}{ll} ?\text{-} & \mathit{solve}([X, \mathit{Andrew}^1]), \mathit{interesting}(\mathit{Juliette}, \mathit{man}, X) \\ & X = \mathit{handsome}^1; \\ & X = \mathit{rich}^1 \end{array}$$

□

Example 7.3 Cooperative answering.

When we put a query to a database system, the system can provide additional information relevant to the query. Cuppens & Demolombe [40] have developed a method based on the use of topics. If the query is a sentence Q, and if Q is about the topic T, then T is identified as a topic of interest, and the system returns, in addition to the answer to Q, other sentences that are about that topic and that are consequences of the database. This idea can be formalised in Reflective Prolog as follows:

$$\begin{array}{l} \mathit{topic}(\mathit{top1},\ \mathit{sentenceA}^1) \\ \mathit{topic}(\mathit{top1},\ \mathit{sentenceB}^1) \\ \mathit{topic}(\mathit{top2},\ \mathit{sentenceC}^1) \\ \mathit{sentenceB} \\ \mathit{sentenceC} \\ \mathit{solve}(Q) \leftarrow \mathit{topic}(T, Q), \mathit{topic}(T, P), P \neq Q, \mathit{solve}(P), \mathit{write}(P) \end{array}$$

$$\begin{array}{l} ?\text{-}\ \mathit{sentenceA} \\ \qquad \mathit{sentenceB}^1 \\ \quad \mathit{yes} \end{array}$$

Cuppens & Demolombe also discuss the possibility of defining a structure on sets of topics. We could for example formalise the fact that a topic *top1* is more specific than a topic *top2* as $\mathit{top1} < \mathit{top2}$. This allows us to represent hierarchies of topics, and it can be used to formalise statements such as: if a topic T_1 is more specific than a topic T_2, and a sentence P is about T_1, then P is also about T_2.

$$\begin{array}{l} \mathit{topic}(\mathit{top1},\ \mathit{sentenceA}^1) \\ \mathit{top1} < \mathit{top2} \\ \mathit{topic}(T_2, P) \leftarrow \mathit{topic}(T_1, P), T_1 < T_2 \end{array}$$

□

7.2 REFLECTIVE PROLOG IN RCL

The language Reflective Prolog can be described in the framework *RCL* as follows. The naming mechanism of the language can be axiomatized as an equality theory along the lines of *NT*, where we have to consider the "concrete" syntax of the language.

RSLD-resolution can be seen as a form of SLD$^{\mathcal{R}}$-resolution which uses reflection axioms implicitly present in the program. Thus, RSLD-resolution can be expressed in *RCL* by two reflection principles, called $\mathcal{U}$ and $\mathcal{D}$. The former corresponds to object-to-meta reflection (case 2 of RSLD-resolution) and the latter corresponds to meta-to-object reflection (case 3 of RSLD-resolution).

The reflection principle $\mathcal{D}$ makes any conclusion drawn at the metaevaluation level available (reflected down) to the base level. Let C be a definite clause.

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

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

- If C takes the form $\mathit{solve}([X, t_1, \ldots, t_n]) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$, then

$$\mathcal{D}(C) = \left\{ \begin{array}{rl|l} p(x_1, \ldots, x_n) \leftarrow & x_1 = \downarrow t_1, \ldots, x_n = \downarrow t_n, & \text{for every } n\text{-ary} \\ & X = p^1, & \text{predicate sym-} \\ & e_1, \ldots, e_q, A_1, \ldots, A_m & \text{bol } p \neq \mathit{solve} \end{array} \right\}.$$

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

$$\mathcal{D}(C) = \left\{ \begin{array}{rl|l} p(y_1, \ldots, y_n) \leftarrow & y_1 = \downarrow X_1, \ldots, y_n = \downarrow X_n, & \text{for every pred-} \\ & X = [p^1, X_1, \ldots, X_n], & \text{icate symbol} \\ & e_1, \ldots, e_q, A_1, \ldots, A_m & p \neq \mathit{solve} \end{array} \right\}.$$

The reflection principle $\mathcal{U}$ makes any conclusion drawn at the base level available (reflected up) to the metaevaluation level.

- If C is of the form $p(t_1, \ldots, t_n) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$, with $p \neq \mathit{solve}$, then

$$\mathcal{U}(C) = \left\{ \begin{array}{rl} \mathit{solve}([p^1, X_1, \ldots, X_n]) \leftarrow & X_1 = \uparrow t_1, \ldots, X_n = \uparrow t_n, \\ & e_1, \ldots, e_q, A_1, \ldots, A_m \end{array} \right\}.$$

RSLD-resolution can then be defined by the following reflection principle $\mathcal{RP}$.

$$\mathcal{RP}(C) = \begin{cases} \mathcal{U}(C) & \text{if } C \text{ is a base level clause,} \\ \mathcal{D}(C) & \text{if } C \text{ is a metaevaluation clause.} \end{cases}$$

Thus, SLD$^{\mathcal{RP}}$-resolution is able to use clauses with conclusion $\mathit{solve}(X)$ to resolve a goal A (meta-to-object reflection), and, vice versa, clauses with conclusion A to resolve a goal $\mathit{solve}(X)$ (object-to-meta reflection).

The following example shows how metaevaluation clauses can play the role of additional clauses for base level predicates.

Example 7.4 Let (P, E) be the following logic program:

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

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

$$\mathcal{RP}(P) = \left\{ \begin{array}{l} p(y,z) \leftarrow \ y = \downarrow Y, z = \downarrow Z, X = p^1, \\ \qquad\qquad \mathit{symmetric}(X), \mathit{solve}([X,Z,Y]) \\ \mathit{solve}([\mathit{symmetric}^1, X]) \leftarrow \ X = \uparrow p^1 \\ \mathit{solve}([p^1, X, Y]) \leftarrow \ X = \uparrow a, Y = \uparrow b \end{array} \right\}.$$

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

$$M_{(P,E)} = \{ \lceil p(a,b) \rceil, \lceil \mathit{symmetric}(p^1) \rceil \}$$

$$M^{\mathcal{RP}}_{(P,E)} = M_{(P,E)} \cup \left\{ \begin{array}{l} \lceil p(b,a) \rceil, \lceil \mathit{solve}([p^1, a^1, b^1]) \rceil, \lceil \mathit{solve}([p^1, b^1, a^1]) \rceil, \\ \lceil \mathit{solve}([\mathit{symmetric}^1, p^2]) \rceil \end{array} \right\}.$$

Thus, by means of $\mathcal{U}$ and $\mathcal{D}$, the first clause of P becomes an axiomatization of symmetry, which can be applied whenever necessary.

The goal $\leftarrow \ p(b,a)$ can be proved from (P, E) by applying the following steps of SLD$^{\mathcal{RP}}$-resolution:

$$\langle \{p(b,a)\}, \{\}, \{\}\rangle$$

$$\langle \left\{ \begin{array}{l} \mathit{symmetric}(p^1), \\ \mathit{solve}([p^1, a^1, b^1]) \end{array} \right\}, \{y = b, Y = b^1, z = a, Z = a^1, X = p^1\}, \{\}\rangle$$

$$\langle \{\mathit{solve}([p^1, a^1, b^1])\}, \{y = b, Y = b^1, z = a, Z = a^1, X = p^1\}, \{\}\rangle$$

$$\langle \{\}, \{y = b, Y = b^1, z = a, Z = a^1, X = p^1\}, \{\}\rangle$$

where the last state is a success state. □

Communication-Based Reasoning

The ability to represent agents and multi-agent cooperation is central to many AI applications. In the context of communication-based reasoning, the interaction among agents is based on communication acts. Agents can, for instance, communicate to manifest their mental state, to announce their goals, to affect the mental state of other agents, and so on. In this chapter we treat the kind of communication where knowledge of agents is passed to other agents.

8.1 COMMUNICATIVE AGENTS

Within the logic programming paradigm, an approach to communication-based reasoning has been proposed by Costantini et al. [34]. In particular, an agent can ask other agents questions in the process of solving a given problem. The main idea of the approach is to represent agents and communication acts by means of theories and reflection principles, respectively. Thus, theories formalise knowledge of agents, while reflection principles characterize possible kinds of interaction among agents.

We use the following definitions.

- ▶ Extend the alphabet of HC^+ to contain a finite set of constants, called **theory symbols**, and the binary predicate symbols *tell* and *told*.

We reserve the characters ω, ϕ and φ for theory symbols.

Below we present the definitions of labelled atom, labelled equation, labelled clause, theory, labelled logic program and labelled query. The intuition is that in order to indicate that a clause C belongs to a theory ω we label C with ω, that is, we write $\omega{:}C$.

- ▶ Let A be an atom, e an equation and ω a theory symbol. Then $\omega{:}A$ is a **labelled atom** and $\omega{:}e$ is a **labelled equation**.

- ▶ Let $\omega_1{:}e_1, \ldots, \omega_q{:}e_q$ $(q \geq 0)$ be labelled equations. Let $\omega{:}A$ and $\omega_{q+1}{:}B_1, \ldots, \omega_{q+r}{:}B_r$ $(r \geq 0)$ be labelled atoms such that $A \neq \mathit{told}(\ldots)$ and $B_i \neq \mathit{tell}(\ldots)$, $1 \leq i \leq r$. Then, the clause

$$\omega{:}A \leftarrow \omega_1{:}e_1, \ldots, \omega_q{:}e_q, \omega_{q+1}{:}B_1, \ldots, \omega_{q+r}{:}B_r$$

  is a **labelled clause**.

Notice that from the definition of labelled clause, the predicate *told* is not definable, while *tell* is, and *tell* cannot appear in the body of any clause.

When a labelled clause takes the form $\omega{:}A \leftarrow \omega{:}e_1, \ldots, \omega{:}e_q, \omega{:}B_1, \ldots, \omega{:}B_r$, we write $\omega{:}\ (A \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r)$ as an abbreviation for it.

- ▶ A **theory** ω is a finite set of labelled clauses of the form:

$$\omega{:}\ (A \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_r).$$

  We call ω the **theory prefix** of that clause.

Theory symbols can appear within a labelled clause $\omega{:}C$ only in referenced form (that is, only the name ϕ^1 of a theory symbol ϕ is allowed in $\omega{:}C$).

- ▶ A **labelled definite program** is a finite set of theories. A **labelled goal** takes the form $\leftarrow \omega_1{:}B_1, \omega_2{:}B_2, \ldots, \omega_r{:}B_r$, where the constituent labelled atoms may have different theory prefixes.

An *agent* is represented by a theory, called the *associated theory* of the agent, and implicitly of the inference mechanism of the language. Thus, declaratively an agent is denoted by the declarative semantics of the associated theory, and procedurally by the closure of the theory itself under resolution.

Communication acts are formalised by means of the predicate symbols *tell* and *told*. They both take as the first argument the name of a theory symbol and as the second argument the name of an expression of the language. Let ω and ϕ be theory symbols and A an atom. The intended meaning of

- $\omega{:}\mathit{tell}(\phi^1, A^1)$ is: the agent ω tells agent ϕ that A, and of
- $\phi{:}\mathit{told}(\omega^1, A^1)$ is: ϕ is told by ω 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. An agent ω may communicate to another agent ϕ everything it can derive (in its associated theory), or only part of what it can derive, or it may even lie, that is, it communicates something it cannot derive.

The intended connection between *tell* and *told* is that an agent ω may use (by means of *told*) only the information that another agent has explicitly addressed to it (by means of *tell*). Thus, an agent can specify, by means of clauses defining the predicate *tell*, its modalities of interaction with the other agents.

What use ϕ makes of this information is entirely up to ϕ. Thus, the way an agent communicates with others is not fixed in the language. Rather, it is possible to define in a program different behaviours for different agents, and/or different behaviours of one agent in different situations as the next examples show. More elaborate examples of the use of agents can be found in [15, 34].

The peculiarity of this approach to theories is that the primitives introduced in the language are not aimed at structuring programs, are not intended as schemas of composition of theories, and are not concerned with logical/ontological modelling of concepts like belief or knowledge. They are simple communication means among theories, on top of which more purpose-oriented mechanisms may be built. In the next section, for example, we provide agents with metalevel reasoning capabilities.

Each agent can specify, by means of clauses defining the predicate *tell*, the modalities of interaction with other agents. These modalities can thus vary with respect to different agents or different conditions. For instance, the following modalities can be expressed.

- An agent ω tells an agent ϕ a thing it can prove but lies about it to an agent φ.

  ω: $\mathit{tell}(\phi^1, p^1) \leftarrow p$
  ω: $\mathit{tell}(\varphi^1,\ \mathit{not}^1\ \ p^1) \leftarrow p$

- An agent ω tells a group of agents whatever it can prove about a predicate p.

  ω: $\mathit{tell}(X, p^1(Y)) \leftarrow Y = \uparrow y, \mathit{group}(X), p(y)$

- An agent ω tells an agent ϕ a thing it can prove within some resource limitations,

ω: *tell*$(\phi^1, X) \leftarrow$ *limited_prove*(X)

where *limited_prove* incorporates the desired limitations.

- An agent ω trusts an agent ϕ but distrusts altogether an agent φ.

  ω: $p(x) \leftarrow x = \downarrow X$, *told*$(\phi^1, p^1(X))$
  ω: $p(x) \leftarrow x = \downarrow X$, *told*$(\varphi^1,$ *not*^{1} $p^1(X))$

The intended connection between *tell* and *told* is formalised by the following reflection principle, called $\mathcal{C}$, mapping labelled clauses into sets of labelled clauses.

• If C is of the form ω: *tell*$(\phi^1, Z) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_n$, then

$$\mathcal{C}(C) = \left\{\phi{:}\textit{told}(\omega^1, Z) \leftarrow \omega{:}e_1, \ldots, \omega{:}e_q, \omega{:}B_1, \ldots, \omega{:}B_n\right\}.$$

• If C is of the form ω: *tell*$(X, Z) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_n$, then

$$\mathcal{C}(C) = \left\{ \begin{array}{rl|l} \varphi{:}\textit{told}(\omega^1, Z) \leftarrow & \omega{:}X = \varphi^1, & \text{for every theory} \\ & \omega{:}e_1, \ldots, \omega{:}e_q, & \text{symbol } \varphi \\ & \omega{:}B_1, \ldots, \omega{:}B_n & \end{array} \right\}.$$

• If C is of the form ω: $B \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_n$, where $B \neq$ *tell*$(\ldots)$, then

$$\mathcal{C}(C) = \{\}.$$

The intuitive meaning is that every time an atom of the form *tell*(ϕ^1, Z) can be derived in a theory ω (which means that agent ω is willing to communicate proposition Z to agent ϕ), the atom *told*(ω^1, Z) can consequently be derived in the theory ϕ (which means that proposition Z becomes available to agent ϕ).

The following example shows how SLD$^{\mathcal{R}}$-resolution works.

Example 8.1 Let E be the equality theory NT and P the following labelled definite program:

ω: *tell*$(\phi^1,$ *ciao*$^1) \leftarrow$ *friend*(ϕ^1)
ω: *friend*(ϕ^1)

ϕ: *hate*(ω^1)

The reflection axioms of P with respect to $\mathcal{C}$ are the following:

$$\mathcal{C}(P) = \left\{ \phi{:}\textit{told}(\omega^1, \textit{ciao}^1) \leftarrow \omega{:}\textit{friend}(\phi^1) \right\}.$$

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

$$M_{(P,E)} = \left\{ \lceil \omega{:}\textit{friend}(\phi^1) \rceil, \lceil \phi{:}\textit{hate}(\omega^1) \rceil, \lceil \omega{:}\textit{tell}(\phi^1, \textit{ciao}^1) \rceil \right\}$$

$$M^{\mathcal{C}}_{(P,E)} = M_{(P,E)} \cup \left\{ \ \lceil \phi{:}\textit{told}(\omega^1, \textit{ciao}^1) \rceil \ \right\}.$$

The labelled goal $\leftarrow \phi{:}\textit{told}(\omega^1, Z)$ can be proved in the following steps.

- The initial state is $S_0 = \langle \{\phi{:}\textit{told}(\omega^1, Z)\}, \{\}, \{\} \rangle$.
- By applying one step of SLD$^{\mathcal{R}}$-resolution, from S_0 and input clause the reflection axiom in $\mathcal{C}(P)$, we obtain the state $S_1 = \langle \{\omega{:}\textit{friend}(\phi^1)\}, \{Z = \textit{ciao}^1\}, \{\} \rangle$.
- Finally, by considering the second labelled clause in the theory ω, we obtain the success state $S_2 = \langle \{\}, \{Z = \textit{ciao}^1\}, \{\} \rangle$.

□

8.2 INTROSPECTIVE, COMMUNICATIVE AGENTS

In this section, we show how to equip communicative agents with metalevel capabilities. Our aim is to build agents that are able to introspect themselves in order to achieve more expressive and reasoning power. We call the introspective, communicative agents **ic-agents**.

To do that, we first define a reflection principle, called $\mathcal{RP}^*$, along the line of $\mathcal{RP}$ used for Reflective Prolog. $\mathcal{RP}^*$ extends $\mathcal{RP}$ to consider labelled clauses. $\mathcal{RP}^*$ is defined as expected. For example, if C is a labelled clause of the form ω: $p(t_1, \ldots, t_n) \leftarrow e_1, \ldots, e_q, A_1, \ldots, A_m$, then

$$\mathcal{RP}^*(C) = \left\{ \begin{array}{rl} \omega\text{: } \textit{solve}([p^1, X_1, \ldots, X_n]) \leftarrow & X_1 = \uparrow t_1, \ldots, X_n = \uparrow t_n, \\ & e_1, \ldots, e_q, A_1, \ldots, A_m \end{array} \right\}.$$

Thus, $\mathcal{RP}^*$ models the connection between the base level and the metaevaluation level within the same theory.

Then, we can define a reflection principle, called $\mathcal{IC}$, based on $\mathcal{RP}^*$ and $\mathcal{C}$ as follows.

$$\mathcal{IC}(C) = \begin{cases} \mathcal{C}(C) & \text{if } C \text{ is a labelled clause of the form} \\ & \omega\text{: } \textit{tell}(\ldots) \leftarrow e_1, \ldots, e_q, B_1, \ldots, B_n, \\ & \\ \mathcal{RP}^*(C) & \text{otherwise.} \end{cases}$$

The reflection principle $\mathcal{IC}$ allows us to model agents that are both communicative (via $\mathcal{C}$) and introspective (via $\mathcal{RP}^*$).

The approach to ic-agents allows us to formalise a variety of interactions among agents depending on the application context. For example, due to the metalevel capabilities of ic-agents, it is possible to generalise the examples of the previous section, like for instance:

The sincere: the agent ω tells all others anything it can prove.

ω: $\mathit{tell}(X,Y) \leftarrow \mathit{solve}(Y)$

The liar: the agent ω lies to the agent ϕ.

ω: $\mathit{tell}(\phi^1,\ \mathit{not}^1\ \ Y) \leftarrow \mathit{solve}(Y)$

The credulous: the agent ω believes whatever is told by the agent ϕ.

ω: $\mathit{solve}(X) \leftarrow \mathit{told}(\phi^1, X)$

The skeptical: the agent ω distrusts altogether the agent ϕ.

ω: $\mathit{solve}(X) \leftarrow \mathit{told}(\phi^1,\ \mathit{not}^1\ \ X)$

The cautious: the agent ω believes any agent Y that it considers reliable.

ω: $\mathit{solve}(X) \leftarrow \mathit{reliable}(Y), \mathit{told}(Y,X)$

This approach to ic-agents has been shown to have enough expressive power for reasoning in non-trivial multi-agent domains, such as the three wise men problem [34]. This formalisation is based on reasoning about communication rather than about beliefs.

We show a simple example of use of ic-agents.

Example 8.2 The following labelled definite program formalises two ic-agents, one of whom is cautious while the other is sincere. The cautious ic-agent ω has partial knowledge about some topic and in the process of proving something it can ask questions to other ic-agents that it considers reliable. The sincere ic-agent ϕ tells a friend everything it can prove.

ω: $\mathit{solve}(X) \leftarrow \mathit{reliable}(Y), \mathit{told}(Y,X)$
ω: $\mathit{reliable}(\phi^1)$
ω: $a \leftarrow b$

ϕ: $\mathit{tell}(X,Y) \leftarrow \mathit{friend}(X), \mathit{solve}(Y)$
ϕ: $\mathit{friend}(\omega^1)$
ϕ: b

The least reflective E-model with respect to $\mathcal{IC}$ is:

$$M^{\mathcal{IC}}_{(P,E)} = \left\{ \begin{array}{l} \lceil \phi{:}solve(b^1)\rceil, \lceil \phi{:}tell(\omega^1, b^1)\rceil, \lceil \omega{:}told(\phi^1, b^1)\rceil, \\ \lceil \omega{:}solve(b^1)\rceil, \lceil \omega{:}b\rceil, \quad \ldots \end{array} \right\}.$$

□

Notice that ic-agents are agents that are communicative and rational. However, an ic-agent cannot offer unsolicited information to some other agent, that is, an ic-agent can tell nothing to an other agent if not explicitly asked (i.e., it is not proactive). In fact, the predicate symbol *tell* cannot appear in the body of any clause. This form of interaction cannot be handled by the framework. This is the main limitation of the approach. If it were possible to do this, that is, an agent ω could autonomously pass information to an agent ϕ, then some form of epistemic updating would have been needed to update the theory underlying agent ϕ.

Chapter 10 introduces a new kind of agent that is aimed to overcome this drawback. In particular, it presents an extension of the approach to rational, reactive agents by Kowalski and Sadri to accomodate introspection and communication among agents.

An Intelligent Tutoring System

Below we sketch the specification of an intelligent tutoring system able to provide students with exercises concerning specific topics they are interested in. The system has its own classification of the users, possibly on the basis of previously selected items.

The agent *interface* defines the user-interface of a system which receives requests for new exercises from the agents corresponding to the users. According to a table that associates students and topics with levels, *interface* asks the agent managing that topic for the exercise, also specifying the level of the user issuing the request.

The agent *math* is an example of an agent managing a topic, i.e., mathematics, that selects an exercise from a suitable library according to the level of expertise of the user. The level (e.g., beginner) and the name of the corresponding library (e.g., *libMB*, i.e., a library of math for beginners) are returned by suitable predicates. The library itself is represented by a predicate, which specifies the description of the exercises. This description is extracted by a predicate *select*, which takes the name of the library as argument. *select* is able to actually access the library by means of a meta-evaluation (*solve*) clause, which commutes from the name to the invocation of the predicate.

Finally, the agent corresponding to each user (for instance, *Dan*) will be able, by means of its metalogic capabilities, to propose the exercise and check the result according to the needs of the particular user. For instance, the text of the exercise can be printed in the language spoken by that user, and can be possibly repeated by using synonyms whenever possible, in case the user does not understand.

Dan: $solve([exercise^1, E]) \leftarrow interested\_in(T),$
$told(interface^1, [exercise\_about^1, T, E])$

$interested\_in(math^1)$
$hobby(football)$
...

interface: $tell(U, [exercise\_about^1, T, E]) \leftarrow level(U, T, L),$
$told(T, [exercise\_level^1, L, E])$

$level(Anne^1, computer\_science^1, intermediate^1)$
$level(George^1, math^1, beginner^1)$
$level(Dan^1, math^1, expert^1)$
...

math: $tell(I, [exercise\_level^1, L, E]) \leftarrow library(L, M), select(E, M)$

$select(E, M) \leftarrow solve([M, E])$

$library(beginner^1, libMB^1)$
$library(intermediate^1, libMI^1)$
$library(expert^1, libME^1)$

$libMB(ex(1, sub(equations), text(\ulcorner x - 3 = 5 \urcorner), res(\ulcorner x = 8 \urcorner)))$
$libMB(ex(2, sub(equations), text(\ulcorner x * x = 16 \urcorner), res(\ulcorner x = 4 \urcorner)))$
...

where $\ulcorner x - 3 = 5 \urcorner$ and $\ulcorner x * x = 16 \urcorner$ above are abbreviations for whatever names are used for these equations.

Now, the user *Dan* can ask the tutoring system for an exercise by asking the query:

$$\leftarrow Dan{:}\ exercise(x).$$

Chapter 9

Plausible Reasoning

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

As a significant example, *replacement based analogy* [122] is a knowledge representation principle which has been shown [39] to be naturally formalisable in terms of reflection.

9.1 REASONING BY ANALOGY

In the context of replacement based analogy, analogy is based on the assumption that if two situations are similar in some respect, then they may be similar in other respects as well. Thus, an analogy is a mapping of knowledge from a known "source" domain into a novel "target" domain. Analogy can be applied to problem-solving, planning, proving, etc., on the basis of the following kind of inference: knowing that from premises A conclusion B follows, and that A' corresponds to A, analogically conclude B'. In particular, replacement-based analogy defines analogy as a replacement of the source object with the target object as stated in the following principle, due to Winston [122].

> *Assume that the premises $\alpha_1, \ldots, \alpha_n$ logically imply α in the source domain. Assume also that analogous premises $\beta_1, \ldots, \beta_n$ hold in the target domain. Then, we conclude the atom β in the target domain which is analogous to α.*

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 [55], or by means of *predicate analogies* and *term correspondence* [39], as examplified below.

Example 9.1 Let P_s and P_t be the following definite programs.

$$P_s \quad \begin{array}{ll} kills(john, george) \leftarrow hates(john, george), has\_weapon(john) & \quad (*) \\ hates(john, george) & \\ has\_weapon(john) & \end{array}$$

$$P_t \quad \begin{array}{l} hates(anne, joe) \\ despises(bill, bill) \leftarrow depressed(bill) \\ has\_weapon(anne) \\ has\_weapon(bill) \\ depressed(bill) \end{array}$$

Let us assume that predicates with the same name are in analogy by default. Let us also assume an explicit analogy between predicates *despises* and *hates*. Clearly, the goal $\leftarrow kills(anne, joe)$ is not provable in P_t. It is however provable by analogy by taking the rule (*) in P_s, and assuming the correspondence between terms $(john, anne)$ and $(george, joe)$. That is, taken the rule (*), we can apply analogy by simulating the corresponding clause of P_t:

$$kills(anne, joe) \leftarrow hates(anne, joe), has\_weapon(anne).$$

Similarly we are able to prove by analogy the goal $\leftarrow kills(bill, bill)$. This is obtained by allowing the correspondence $(john, bill)$ and $(george, bill)$. □

In a term correspondence, an element of the source domain can correspond to only one element of the target domain; on the contrary, two distinct elements of the source domain, like the constants *john* and *george* in the example above, may correspond to the same element of the target domain. Thus, a term correspondence can be seen as a (non injective) function.

The reflective semantics of this kind of analogical reasoning can be defined in *RCL* as follows. We consider logic programs of the form (P, E), where $P = P_s \cup P_t$. Since we want to distinguish the clauses in P_s form the ones in P_t, we can label every clause to indicate whether it belongs to P_s or P_t. Let $pred(P_s)$ and $pred(P_t)$ be the sets of predicate symbols of P_s and P_t, respectively.

▶ Let $(P_s \cup P_t, E)$ be a logic program. A **predicate analogy** is a finite (possibly, empty) set of pairs (p, q) such that $p \in pred(P_s)$ and $q \in pred(P_t)$.

▶ Let $s_1, \ldots, s_n$ be terms of the language of P_s and $t_1, \ldots, t_n$ terms of the language of P_t. A **term correspondence** is a finite (possibly, empty) set of pairs of terms, written as $\{s_1 // t_1, \ldots, s_n // t_n\}$, such that, for every i and j, with $1 \leq i, j \leq n$, the following conditions hold:

- $s_i \not\equiv s_j$, when $i \neq j$,
- s_i is not a subterm of s_j, when $i \neq j$, and
- $\mathit{vars}(s_i) \cap \mathit{vars}(t_j) = \{\}$.

Clearly, substitutions are particular cases of correspondences. A correspondence $\sigma = \{s_1 // t_1, \ldots, s_n // t_n\}$ can be applied to a term or atom A by replacing every occurrence of s_i in A with t_i. The result of the application is indicated by $A\sigma$. Term correspondences can be composed with substitutions, giving a new term correspondence as a result. In particular, given a substitution θ, the term correspondence $\sigma' = \sigma\theta$ is obtained by substituting every variable in σ with its assignment in θ, if any.

The semantics of this kind of analogical reasoning can be expressed in terms of a reflection principle, called $\mathcal{A}$, defined below. Given a predicate analogy S and a term correspondence σ, define a relation r over clauses as follows:

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 the reflection principle $\mathcal{A}$ mapping clauses of P_s into clauses of P_t.

• Given a predicate analogy S, a term correspondence σ and a clause C in P_s, then

$$\mathcal{A}(C) = \{D \mid \text{for all } D \text{ such that } r(C, D) \text{ holds}\}.$$

By applying the previous definition, the mapping $T^{\mathcal{A}}_{(P,E)}$, which allows the derivation of analogical consequences, characterizes the consequences of P_t with respect to the clauses of P_t itself and the clauses of P_s.

Example 9.2 Let E be an equality theory and $P = P_s \cup P_t$ the program of Example 9.1. Assume that:

$$\sigma = \{\mathit{john} // \mathit{bill}, \mathit{george} // \mathit{bill}\}$$

$$S = \{(\mathit{hates}, \mathit{despises}), (\mathit{kills}, \mathit{kills}), (\mathit{has\_weapon}, \mathit{has\_weapon})\}.$$

Then, the least reflective E-model $M^{\mathcal{A}}_{(P,E)}$ of (P, E) contains $\mathit{kills}(\mathit{bill}, \mathit{bill})$ in P_t. In fact, by applying the reflection principle $\mathcal{A}$ to the clause:

$$\mathit{kills}(\mathit{john}, \mathit{george}) \leftarrow \mathit{hates}(\mathit{john}, \mathit{george}), \mathit{has\_weapon}(\mathit{john})$$

in P_s, we obtain in P_t the clause:

$$\mathit{kills}(\mathit{bill}, \mathit{bill}) \leftarrow \mathit{despises}(\mathit{bill}, \mathit{bill}), \mathit{has\_weapon}(\mathit{bill}).$$

It is easy to see that the goal $\leftarrow \mathit{kills}(\mathit{bill}, \mathit{bill})$ is provable in P_t. $\square$

Multi-agent Systems

As the size and complexity of software systems becomes greater, a fundamental shift of paradigm is required from the current software engineering approaches. One of the new approaches is *distributed artificial intelligence* (DAI). DAI is a subfield of *artificial intelligence* (AI) that aims to construct systems composed of multiple problem solving entities which interact with each other to enhance their performance.

DAI has two subfields that are strongly interrelated: *distributed problem solving* (DPS) and *multi-agent systems* (MAS). Whereas research in DPS considers how the work of solving a particular problem can be divided among a number of cooperating modules, research in MAS is concerned with coordinating intelligent behaviour among a collection of agents, how agents can coordinate their plans to take an action. Similarly to modules in a DPS system, agents share knowledge about problems and solutions, but to coordinate their actions, agents need to represent and reason about the knowledge, actions and plans of other agents.

In this chapter, we focus on multi-agent systems and introduce basic notions and terminology needed to read the next chapter. We present a classification of agents and illustrate some of the main proposed approaches. We begin with a survey of some definitions of agents appearing in the literature.

10.1 WHAT IS AN AGENT ?

Although the term "agent" is now very popular and widely used in many closely related areas, there is no a single, uniform definition of agenthood in the literature. In AI the original sense of the term was of someone acting on behalf of someone else; now people often use it to refer to an entity that functions continuously and autonomously in some environment where other processes exist. Several definitions of agent have been proposed (for a survey see, e.g., [49]), including the followings.

Russell & Norvig agents [99, p. 33]. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment.

Maes agents [89, p. 108]. Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so, realize a set of goals or tasks for which they were designed.

IBM agents [61]. Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in doing so, employ some knowledge or representation of the user's goals or desires.

Wooldridge-Jennings agents [123, p. 2]. "...a hardware or (more usually) software-based computer system that enjoys the following properties:

- autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- social ability: agents interact with other agents (and possibly humans) via some kind of agent communication language;
- reactivity: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined) and respond in a timely fashion to changes that occur in it;
- proactiveness: "...agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative."

Genesereth-Ketchpel agents [54, p. 48]. "...software components that communicate with their peers by exchanging messages in an expressive agent communication language. Agents can be as simple as subroutines; but typically they are larger entities with some sort of persistent control."

Shoham agents [103, p. 52]. "An agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices and commitments.... In this view, therefore, agenthood is in the mind of the programmer. What makes any hardware or software component an agent is precisely the fact that one has chosen to analyse and control it in these mental terms. The question of what an agent is is now replaced by the question of what entities can be viewed as having a *mental state*."

However, there are several widely accepted concepts which characterize agent systems. The next section places existing agents into various agent classes along the lines of Nwana and Ndumu [86, 87].

10.2 TYPOLOGY OF AGENTS

There are several dimensions when classifing software agents:

- Agents may be classified as being either *rational* (also referred as *deliberative*) or *reactive*. Typically, rational agents have a model of the environment and possess knowledge about the effects of their actions. Typically, they are also capable of planning in order to achieve their goals. In contrast, reactive agents have no world model and they react to the environment by using a stimulus/response type of behaviour.
- Agents may be classified along several attributes they may possess, for example:
  - autonomy: they can operate on their own without any human intervention;
  - cooperation: they have the ability to interact with other agents;
  - learning: they are able to change their behaviour from previous experience;
- Agents may be classified by their mobility, i.e., by their ability to move around some network. This yields the classes of *static* and *mobile* agents.
- Other attributes of agents can also be considered. For example, an agent may be *benevolent* or *non-helpful*, *antagonistic* or *altruistic*, it may *lie* or it is always *truthful*, and so on. (See, for instance, how some of the properties above can be formalised within the framework of ic-agents (Section 8.2).)

Agents may be classified according to which subset of these properties they enjoy. For instance, we can have static, deliberative agents, or mobile, reactive agents, and so on. Hence, a hierarchical classification of agents may be based on set inclusion, e.g., static, deliberative agents are a subclass of static agents.

10.3 OBJECT-ORIENTED VS. AGENT-ORIENTED PROGRAMMING

The *object-oriented programming* (OOP) paradigm proposes viewing a computational system as composed of "objects" that interact with each other in terms of sending and receiving messages. An object is a module that is composed of data and procedures. The behaviour required from an object by other objects may be triggered by invoking the object-interaction mechanism, which is usually implemented through some kind of message-passing.

Typically, objects in OOP are "passive" entities in the sense that they show no activity without being triggered by explicit messages from other objects. In contrast, if they are not passive in this sense then they are commonly called "agents" which may be viewed as processes, but with an object-oriented structure.

The term *agent-oriented programming* (AOP) was introduced by Shoham [103] to denote a computational framework which can be seen as a specialization of the OOP paradigm. He defined an AOP system so as to include:

- a formal language for describing mental states[1];
- a programming language in which to define and program agents; the semantics of the programming language has to be faithful to the semantics of the mental state.

Table 1 describes the relation between AOP and OOP. AOP specialises OOP by fixing the state (now called mental state) of the objects (now called agents) to consist of components such as beliefs, capabilities, and so on. A computation in AOP consists of such agents informing, requesting, assisting, etc. Shoham borrowed this idea from *speech act* literature. In fact, speech act theory categorises different types of communicative acts, such as informing, requesting, and so on, and considers different presuppositions and effects of communicative acts.

	OOP	*AOP*
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	commitments, beliefs, ...
Process of computation	message passing response methods	message passing and response methods
Types of messages	unconstrained	inform, request, offer, ...
Constraints on methods	none	honesty, consistency, ...

Table 1 – OOP vs. AOP

[1]See the definition of Shoham agents, p. 102.

The central problem in AOP is how to achieve coordinated action among agents, so that they can accomplish more as a group than each of them can individually.

In contrast to Shoham's position, Genesereth & Ketchpel [54] argue that there is a key distinction between OOP and AOP: in OOP the meaning of a message may differ from object to object (this is the principle of polymorphism); in AOP agents use a common language with an agent-independent semantics. Genesereth & Ketchpel highlight three important questions raised by the new AOP paradigm. They include [54, p. 48]:

- What is an appropriate agent communication language?
- How are agents capable of communicating in this language?
- What communication architectures promote cooperation?

They begin addressing such issues via an agent communication language called ACL.

10.4 REACTIVE AGENTS

A *reactive* agent is an agent that acts in a stimulus-response manner to the present state of the environment in which it is embedded. It does not necessarily possess any symbolic representation of its environment (the world in which the agent is situated serves as its own model) and does not perform any symbolic reasoning. It has no explicit goal. (For an overview on reactive agents see, e.g., [86, 123].)

Reactive agents are relatively simple, easy to understand and interact with other agents in basic ways. They are situated, i.e., they do not plan ahead or revise any world models, and their actions depend on what happens at the present moment. However, complex patterns of behaviour emerge from these interactions when agents are viewed globally.

A reactive agent can be viewed as executing condition-action rules.

Example 10.1 The behaviour of a reactive agent may be regulated by the following condition-action rule:

if there is an intruder **then** raise an alarm.

Such a rule is triggered by an observation of an intruder and generates an action of raising an alarm as output. Note that the agent does not have any explicit goal or belief. This rule achieves a goal, i.e., maintaining security, which is implicit rather than explicit. □

Works on reactive agents date back to the works of Brooks in the middle '80s. Brooks heavily criticised the symbolist tradition of AI and proposed three key theses [23]:

1. Intelligent behaviour can be generated *without explicit representation* of the kind that symbolic AI proposes.
2. Intelligent behaviour can be generated *without explicit reasoning* of the kind that symbolic AI proposes.
3. Intelligence is an *emergent* property of certain complex systems.

Brooks argued that these theses obviate the need for symbolic representations or models because the world becomes its own best model. Furthermore, this model is always kept up-to-date since the system is connected to the world via sensors and/or actuators.

Brooks identified two key ideas that have characterized his research:

1. Real intelligence is situated in the world, not in systems like theorem provers or expert systems.
2. Intelligent behaviour arises as a result of an agent's interaction with its environment.

In order to demostrate his claims, he built a number of robots based on an abstract architecture, called the *subsumption architecture* [22]. This architecture consists of a set of modules, each of which is based on a finite state machine. Every finite state machine is triggered into action when its input signal exceeds some threshold. Finite state machines represent the only processing units in this architecture. In it there is no facility for global control and no means for accessing global data.

The modules are grouped and placed into layers that connect sensing to acting and run in parallel. Lower level layers allow the agent to react to important or dangerous events, while modules in a higher level can inhibit modules in lower layers. Each layer has a hard-wired purpose or behaviour, e.g., to avoid obstacles or to enable/control wandering. In this architecture it is possible to add new functionality by adding new, higher level layers.

Brooks' subsumption architecture

10.5 RATIONAL/DELIBERATIVE AGENTS

Typically, the notion of a *rational* agent in AI focuses on the thinking process of the agent and ignores its interaction with the environment. Following Wooldridge & Jennings [123], a rational agent can be seen as an agent which contains an explicitly represented, symbolic model of the world, and in which decisions are made via logical reasoning based on pattern matching and symbolic manipulation. For instance, ic-agents (presented in Section 8.2) can be classified as rational agents.

Kowalski & Sadri [71] outline an abstract procedure which defines the observation-thought-action cycle of rational agents. They express the cycle at the top-most level as follows.

To cycle at time t,

(*i*) observe any input at time t,

(*ii*) record any such input,

(*iii*) check the inputs for satisfaction of integrity constraints by reasoning forwards from the input,

(*iv*) solve goals by constructing a plan, using for steps (*iii*) and (*iv*) a total of r units of time,

(*v*) select a plan from among the alternatives, and select from the plan an atomic action which can be executed at time $t + r + 2$,

(*vi*) execute the selected action at time $t + r + 2$ and record the result,

(*vii*) cycle at time $t + r + 3$.

The cycle starts at time t by observing and recording any inputs from the environment (steps (*i*) and (*ii*)). Time t is the clock of the agent. Steps (*i*) and (*ii*) are assumed to take one unit of time each. Steps (*iii*) and (*iv*) conjoined are assumed to take r units of time. The amount r of resources that an agent can spend on "thinking" in steps (*iii*) and (*iv*) is potentially *unbounded.* Note that only after having generated a *complete* plan (step (*iv*)), the agent begins to execute it (step (*vi*)). Steps (*v*) and (*vi*) conjoined are assumed to take one unit of time.

A well known architecture for rational agents is the BDI-architecture proposed by Rao & Georgeff [96]. They characterize a rational agent in terms of the mental attitudes of *beliefs*, *desires* and *intentions.* These attitudes denote the following states of the agent:

- beliefs can be seen as the informative state of the agent,
- desires, or goals (the objectives to be accomplished), can be seen as the motivational state of the agent,
- intentions represent the decisions the agent has made previously to commit itself to any given action.

Rao & Georgeff model these mental attitudes as modal operators in a logic and state some interesting requirements on their semantics. The representation language adopted for the BDI-architecture is a modal language based on an adaptation of "computation tree logic", and is used as a meta-language for depicting the mental states of the agent. A Kripke "possible world" semantics with an extension of temporality is assumed for the language. Each possible world is a temporal structure with a single past and branching future.

An abstract interpreter for BDI-architecture can be outlined as follows. Let B, D and I be a set of beliefs, desires and intentions, respectively.

```
BDI-interpreter
repeat
    options ← option_generator(event_queue,B,D,I)
    selected_options ← deliberate(options,B,D,I)
    update_intentions(selected_options,I)
    execute(I)
    get_new_external_events
    drop_successful_attitudes(B,D,I)
    drop_impossible_attitudes(B,D,I)
end repeat
```

At the beginning of every cycle, the interpreter reads the event_queue and returns a list of options. Then, the deliberator selects the options to be adopted and adds them to I. If I contains an atomic action, then the agent may execute it. At this stage in the cycle, any external events (i.e., observations) which may have occurred are added to the event queue. Finally, the agent drops all successful desires and intentions as well impossible ones, and cycles again.

This agent-oriented approach has been adopted in several applications and it has been proved suitable for building complex distributed systems [97].

The rational approach to agents has two major problems:

1. the problem of translating the world into an adequate symbolic description, and
2. the problem of how to represent information about the world in such a way that agents can reason with it in an acceptable fixed time bound.

Brooks [23], for example, argues that the lack of a bound on r makes the approach unfeasible: a rational agent is not able to react appropriately and in real time to the changes in its environment. This criticism has lead to the development of a kind of agent that is both rational and reactive. This is the topic of the next section.

10.6 HYBRID AGENTS

A *hybrid* agent is an agent that incorporates the characteristics of two or more different types of agents with the aim to bring together the benefits of both.

Frequently, hybrid agents are based on the combination of reactive agents, which are capable of reacting to events that occur in the environment, and rational agents, which are capable of developing plans and make decisions.

This kind of structuring is employed for example by Kowalski & Sadri. They propose a *unified architecture* [71] (outlined in Section 11.4) which combines rationality with reactivity. This architecture employs a proof procedure [52], called the *IFF procedure*, as the "thinking" component of the agent. The procedure combines definitions with integrity constraints. It uses definitions for "rational" reduction of goals to subgoals and integrity constraints for reactive, condition-action rule behaviour. Furthermore, they also allow the proof procedure to be interrupted (by making it resourse-bounded) in order to assimilate observations from the environment and performing actions.

Chapter 11

ICRR Agents

We propose an approach to model logic-based agents that can reason about their own beliefs as well as the beliefs of other agents, and can communicate with each other. The agents can be reactive, rational or hybrid, combining both reactive and rational behaviour.

11.1 MOTIVATION

Kowalski & Sadri [71] propose an approach to agents within an extended logic programming framework. We will refer to these agents as **rr-agents**. rr-agents are *hybrid* in that they exhibit both *rational* (or *deliberative*) and *reactive* behaviour. The reasoning core of rr-agents is a proof procedure that combines forward and backward reasoning. Backward reasoning is used primarily for planning, problem solving and other deliberative activities. Forward reasoning is used primarily for reactivity to the environment, possibly including other agents. The proof procedure is executed within an observe-think-act cycle that allows the agent to be alert to the environment and react to it as well as think and devise plans. Both the proof procedure and the rr-agent architecture can deal with temporal information. The proof procedure (IFF proof procedure [52]) treats both inputs from the environment and agents' actions as *abducibles* (hypotheses).

Chapter 8 presents an approach within the *RCL* programming paradigm to model rational agents that are introspective and that communicate with each other. In that approach introspection is achieved via the meta-predicate *solve* and communication is achieved via the meta-predicates *tell* and *told.* A communication act is triggered everytime an agent ω has a goal of the form

$$\omega : \mathit{told}(\phi, A).$$

This stands both for "ω is told by ϕ that A" as well as "ω asks ϕ whether A". This goal is solved by agent ϕ telling ω that A, that is, $\phi : \mathit{tell}(\omega, A)$, standing for "$\phi$ tells ω that A". Thus, agent ϕ is explicitly asked about A. The information is passed from ϕ to ω by eventually instantiating A.

A main limitation of this approach is that agents cannot tell anything to other agents unless explicitly asked.

The ability to provide agents with some sort of "proactive" communication primitive is widely discussed in literature [32, 75, 104, 118, 119, 120]. For example, one can model agents that advertise their services so that other agents, possibly with the help of *mediators*, can find agents that provide services for them.

An interesting application of agents is that of a virtual marketplace on the Web where users create autonomous agents to buy and sell goods on their behalf. Chavez & Maes [28], for example, propose a marketplace, where users can create selling and buying agents by giving them a description of the item they want to sell (or to buy). The main goal of the approach is to help users in the negotiations between buyers and sellers, and to sell the goods better (i.e., at a higher price) than the user would be able to, by taking advantage of their processing speed and communication bandwidth. Chavez & Maes' agents are:

(*i*) proactive: "...they try to sell themselves, by going into a marketplace, contacting interested parties (namely, buying agents) and negotiating with them to find the best deal", and

(*ii*) autonomous: "...once released into the marketplace, they negotiate and make decisions on their own, without requiring user intervention".

Chavez & Maes point out their agents' lack of rationality: "Our experiment demonstrated the need and desire for 'smarter' agents whose decision-making processes more closely mimic those of people and which can be directed at a more abstract, motivational level."

In this chapter, we propose a combination of (a version of) ic-agents (introduced in Section 8.2) and rr-agents. In the resulting framework reactive, rational or hybrid agents can reason about their own beliefs as well as about the beliefs of other agents and they can communicate proactively with each other. In the framework, the agents' behaviour can be regulated by condition-action rules such as: If I am asked by another, friendly agent about something and I can prove it from my beliefs, then I will tell the agent about it.

In the proposed approach, the two primitives for communication of ic-agents, *tell* and *told*, are seen as actions within the cycle of the rr-agent architecture and therefore they are treated as abducibles.

As we model communication acts between agents by a form of metalevel communication, the epistemic effects of communication may easily be assimilated in the knowledge base of the agent, if so desired, via metalevel predicates.

The approaches to ic-agents and rr-agents being orthogonal, they can easily be integrated. The resulting framework for agents, which we call ***icrr-agents***, seems to be promising and powerful enough to solve several problems in AI. In Section 11.9, we show how several kinds of models of communication can be treated in such a framework.

11.2 PRELIMINARIES

Let L be a metalanguage. Assume that the alphabet of L includes the predicate symbols $=$, *true*, *false*, *solve*(X), *tell*(X,Y) and *told*(X,Y).

Notation. We write a vector of terms $t_1,\ldots,t_n$ as $\vec{t}$, and abbreviate $x_1 = t_1,\ldots,x_n = t_n$ as $\vec{x} = \vec{t}$. We use lower-case characters for variables and upper-case characters for metavariables. We use the characters ω, ϕ and φ for agents. □

- ▶ A **logic program** is a set of clauses of the form:

$$A \leftarrow L_1 \wedge \ldots \wedge L_n \qquad (n \geq 0)$$

where every L_i, $1 \leq i \leq n$, is a literal and A is an atom whose predicate symbol is different from $=$, *true* and *false*. If $A = p(\vec{t})$, the clause is said to **define** p.

All variables in any clause of a logic program are implicitly universally quantified. The IFF proof procedure relies upon the completion of logic programs [29].

- ▶ The **completion of a predicate** p is defined, for a logic program P given by the set of clauses:

$$\begin{array}{l} p(\vec{t_1}) \leftarrow D_1 \\ \quad\vdots \\ p(\vec{t_k}) \leftarrow D_k \quad (k \geq 1) \end{array}$$

as the **iff-definition**:

$$p(\vec{x}) \leftrightarrow [\vec{x} = \vec{t_1} \wedge D_1] \vee \ldots \vee [\vec{x} = \vec{t_k} \wedge D_k].$$

If p is not defined in P, then the completion of p is the iff-definition:

$$p(\vec{x}) \leftrightarrow \mathit{false}.$$

Given an iff-definition of the form $p(x_1, \ldots, x_n) \leftrightarrow D_1 \vee \ldots \vee D_n$, the variables $x_1, \ldots x_n$ are implicitly universally quantified, with the scope being the entire definition. Any variable in a disjunct D_i which is not one of $x_1, \ldots, x_n$ is implicitly existentially quantified, with the scope being the entire disjunct.

▶ The **selective completion** $comp_S(P)$ of a logic program P with respect to a set S of predicates of L is the union of the completions of all predicates in S.

▶ The **completion** $comp(P)$ **of a logic program** P is the union of the completion of all predicates of L.

Example 11.1 Let L contain the predicate symbols a, b, c, d and $solve$ and let $S = \{a, c, d, solve\}$. Let P be the following logic program:

$$P = \left\{ \begin{array}{l} solve(a^1) \leftarrow b \\ c \end{array} \right\}.$$

Then, the selective completion of P with respect to S is:

$$comp_S(P) = \left\{ \begin{array}{l} a \leftrightarrow false \\ c \leftrightarrow true \\ d \leftrightarrow false \\ solve(X) \leftrightarrow [X = a^1 \wedge b] \end{array} \right\}.$$

□

▶ An **integrity constraint** is an implication of the form:

$$L_1 \wedge \ldots \wedge L_n \Rightarrow A \qquad (n \geq 0)$$

where $L_1, \ldots, L_n$ are literals, possibly *true*, and A is an atom, possibly *false*. When $n = 0$, the conjunction is equivalent to *true*. A is called the *head* and $L_1 \wedge \ldots \wedge L_n$ the *body* of the integrity constraint.

All variables in an integrity constraint are implicitly universally quantified. For notational convenience we abbreviate integrity contraints of the form $true \Rightarrow A$ as A and we use the inequality $s \neq t$ as an abbreviation for $s = t \Rightarrow false$.

▶ An **abductive logic program** [66] is a tuple $\langle P, \mathcal{A}, I \rangle$, where P is a logic program, $\mathcal{A}$ a set of predicates, and I a set of integrity constraints such that:

1. $=$, *true*, *false* and *solve* do not occur in $\mathcal{A}$, and
2. $p \in \mathcal{A}$ implies that p is not defined in P.

The predicates in $\mathcal{A}$ are referred as **abducible** and the atoms built from the abducible predicates are referred to as **abducible atoms**.

Assuming that abducible predicates have no definitions in P does not result in a loss of generality (see Kakas et al. [67]). Abducibles can be thought of as hypotheses that can be used to extend the given logic program in order to provide an "explanation" for given queries (or observations), that "satisfy" the integrity constraints. Different notions of "explanation" and "satisfaction" have been used in the literature. The simplest notion of "satisfaction" is consistency of the explanation with the program and the integrity constraints.

In the rest of this chapter, $\overline{\mathcal{A}}$ will refer to the complement of the set $\mathcal{A}$ with respect to the set of all predicates in L, i.e., the set of non-abducible predicates.

Example 11.2 Let the abductive logic program $\langle P, \mathcal{A}, I\rangle$ be defined as follows:

$$\begin{aligned}
P &= \left\{ \begin{array}{l} has(x,y) \leftarrow buys(x,y) \\ has(x,y) \leftarrow steals(x,y) \\ honest(Tom) \end{array} \right\} \\
\mathcal{A} &= \{\ buys, steals\ \} \\
I &= \{\ honest(x) \wedge steals(x,y) \Rightarrow false\ \}.
\end{aligned}$$

Then, given the observation $G = has(Tom, computer)$, the set of abducibles $\{buys(Tom, computer)\}$ is an explanation for G, satisfying I, whereas the set $\{steals(Tom, computer)\}$ is not, because it is inconsistent with I. □

11.3 THE IFF PROOF PROCEDURE

The IFF proof procedure [52] is a rewriting procedure, consisting of a number of inference rules, each of which replaces a formula by one which is equivalent to it in a theory T of iff-definitions defined as:

$$T = comp_{\overline{\mathcal{A}}}(P),$$

for a given abductive logic program $\langle P, \mathcal{A}, I\rangle$. The inference rules are the following. [1]

[1] The full IFF proof procedure includes two additional inference rules: case analysis and factoring. Here, we omit these rules for simplicity.

Unfolding: given an atom $p(\vec{t}\,)$ (p possibly being *solve*) and an iff-definition
$p(\vec{x}\,) \leftrightarrow D_1 \vee \ldots \vee D_n$ in T $(n > 0)$,
$p(\vec{t}\,)$ is replaced by
$(D_1 \vee \ldots \vee D_n)\theta$, where θ is the substitution $\{\vec{x}\,/\vec{t}\,\}$.

Propagation: given an atom $p(\vec{s}\,)$ and an integrity constraint
$L_1 \wedge \ldots \wedge p(\vec{t}\,) \wedge \ldots \wedge L_n \Rightarrow A$,
a new integrity constraint
$L_1 \wedge \ldots \wedge \vec{t} = \vec{s} \wedge \ldots \wedge L_n \Rightarrow A$
is added.

Logical simplification:
$[B \vee C] \wedge E$ is replaced by $[B \wedge E] \vee [C \wedge E]$ (**splitting**),
$\mathit{not}\, A \wedge B \Rightarrow C$ is replaced by $B \Rightarrow C \vee A$ (**negation elimination**),
$B \wedge \mathit{false}$ is replaced by *false*,
$B \wedge \mathit{true}$ is replaced by B,
$B \vee \mathit{false}$ is replaced by B,
$B \vee \mathit{true}$ is replaced by *true*.

Equality rewriting: applies the equality rewrite rules, simulating the E-unification algorithm, and the application of substitutions.

The following example shows equality rewriting for the Clark's equality theory.

Example 11.3 The following rules, proposed by Fung & Kowalski [52], are the equality rewrite rules for CET, the Clark's equality theory [29].

They are applied both to an equality that occurs as a conjunct and to an equality that occurs in the body of an integrity constraint.

1. Replace $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ by $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$.
2. Replace $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ by *false* whenever f and g are distinct.
3. Replace $t = t$ by *true*, for any term t.
4. Replace $x = t$ by *false* whenever t is a term containing the variable x.

5a. Replace $t = x$ by $x = t$ whenever x is a variable and t is not.

5b. Replace $y = x$ by $x = y$ whenever x is a universally quantified variable and y is not.

6a. If $x = t$ occurs as a conjunct and x does not occur in t, then apply the substitution x/t to the entire conjunction, retaining the conjunct $x = t$ intact.

6b. If $x = t$ occurs in the body of an integrity constraint, the variable x does not occur in t and x is universally quantified, then apply the substitution x/t to the integrity constraint, deleting the equality.

□

Remark. If L is a metalanguage containing the operators $\uparrow$ and $\downarrow$ to compute names of expressions of L, then *CET* has to be extended accordingly. For example, if L is the metalanguage HC^{+}, we can extend *CET* with the equality theory *NT*. Thus, *CET* consists of equations formalising equality between terms in normal form and *NT* consists of name equations formalising equality between terms containing $\uparrow$ and $\downarrow$. This extension is possible because *NT* is consistent. Furthermore, *NT* being sufficiently complete, for any ground terms s and t, we have that

$$CET \cup NT \models s = t \quad \text{iff} \quad CET \models \lceil s \rceil = \lceil t \rceil .$$

With respect to $CET \cup NT$ the equality rewrite rules of Example 11.3 need to be modified (e.g., they can be adapted from the E-unification algorithm *UN*). □

Now, we introduce the notions of an initial goal G and of a derivation for G.

▶ An **initial goal** is a conjunction of literals whose variables are free.

Let N be any disjunct N_i $(1 \leq i \leq m)$ occurring in a given disjunction $N_1 \vee \ldots \vee N_m$. Then, we write *Rest* to indicate the disjunction $N_1 \vee \ldots \vee N_{i-1} \vee N_{i+1} \vee \ldots \vee N_m$. The notation $N \vee REST$ says that N is the "selected" disjunct and *Rest* is the disjunction consisting of the remaining disjuncts of the given disjunction.

▶ Given an abductive logic program $\langle P, \mathcal{A}, I \rangle$ and an initial goal G, a **derivation** for G is a sequence of formulae

$$\begin{array}{rcl} F_1 & = & G \wedge I \\ & \vdots & \\ F_n & = & N \vee Rest \end{array}$$

such that each **derived goal** F_{i+1} in the sequence is obtained from F_i by applying one of the inference rules, as follows:

– Unfolding: applied to iff-definitions and either (i) to atoms in conjuncts in F_i or (ii) to atoms in bodies of integrity constraints in F_i,

- Propagation: applied to atoms that are conjuncts in F_i and integrity constraints in F_i,
- Logical simplification[2]: applied to formulae,
- Equality rewriting: applied to equalities.

Every negative literal $\mathit{not} A$ as a conjunct in the initial goal as well as in any derived goal is rewritten as an integrity constraint $A \Rightarrow \mathit{false}$.

Every derivation relies upon some control strategy. Some strategies are preferable to others. E.g., splitting should always be postponed, because it is an explosive operation.

▶ Let $\langle P, \mathcal{A}, I\rangle$ be an abductive logic program, G an initial goal and $F_1 = G \wedge I, \dots, F_n = N \vee \mathit{Rest}$ a derivation for G. If $N \neq \mathit{false}$, N is some conjunction of literals and integrity constraints, and no inference rule can be applied to N, then $F_1, \dots, F_n$ is a **successful derivation**.

▶ Let $\langle P, \mathcal{A}, I\rangle$ be an abductive logic program, G an initial goal and $F_1 = G \wedge I, \dots, F_n = N \vee \mathit{Rest}$ a successful derivation for G. An **answer extracted from** N is a pair (D, σ) such that:

- σ' is a substitution replacing all free and existentially quantified variables in N by ground terms in N and σ' satisfies all equalities and inequalities in N,
- D is the set of all abducible atoms that are conjuncts in $N\sigma'$ and σ is the restriction of σ' to the variables in G.

Observe that every abducible atom in D is ground.

Example 11.4 Let $\langle P, \mathcal{A}, I\rangle$ be the abductive logic program in Example 11.2. Then, $\mathit{comp}_{\overline{\mathcal{A}}}(P)$ is

$$\left\{ \begin{array}{l} \mathit{has}(x,y) \leftrightarrow \mathit{buys}(x,y) \vee \mathit{steals}(x,y) \\ \mathit{honest}(x) \leftrightarrow x = \mathit{Tom} \end{array} \right\}.$$

The following is a (successful) derivation for the goal $G = \mathit{has}(\mathit{Tom}, \mathit{computer})$. Note that the variables x and y in the derivation are universally quantified because they belong to an integrity constraint.

[2]To simplify the treatment of quantifiers, both iff-definitions and integrity constraints are assumed to be range-restricted, i.e., all variables in the head must appear in at least one atom in the body, and this atom must not be an equality between two variables.

$$\begin{aligned}
F_1 &= G \wedge I \\
&\quad \text{(by unfolding)} \\
F_2 &= G \wedge [x = \mathit{Tom} \wedge \mathit{steals}(x, y) \Rightarrow \mathit{false}] \\
&\quad \text{(by equality rewriting–6b)} \\
F_3 &= G \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}] \\
&\quad \text{(by unfolding)} \\
F_4 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \vee \mathit{steals}(\mathit{Tom}, \mathit{computer})] \wedge \\
&\quad [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}] \\
&\quad \text{(by splitting)} \\
F_5 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]] \vee \\
&\quad [\mathit{steals}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]] \\
&\quad \text{(by propagation)} \\
F_6 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]] \vee \\
&\quad [\mathit{steals}(\mathit{Tom}, \mathit{computer}) \wedge [y = \mathit{computer} \Rightarrow \mathit{false}]] \\
&\quad \text{(by equality rewriting–6b)} \\
F_7 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]] \vee \\
&\quad [\mathit{steals}(\mathit{Tom}, \mathit{computer}) \wedge \mathit{false}] \\
&\quad \text{(by logical simplification)} \\
F_8 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]] \;\vee\; \mathit{false} \\
&\quad \text{(by logical simplification)} \\
F_9 &= [\mathit{buys}(\mathit{Tom}, \mathit{computer}) \wedge [\mathit{steals}(\mathit{Tom}, y) \Rightarrow \mathit{false}]].
\end{aligned}$$

The answer extracted from F_9 is $(\{\mathit{buys}(\mathit{Tom}, \mathit{computer})\}, \{\})$. In fact, we have that $\sigma' = \{\}$ because the only variable occurring in F_9 (i.e., y) is universally quantified, and so $\sigma = \{\}$. By noting that the only abducible atom occurring as a conjunct in F_9 is $\mathit{buys}(\mathit{Tom}, \mathit{computer})$, it follows that $D = \{\mathit{buys}(\mathit{Tom}, \mathit{computer})\}$. □

11.4 KOWALSKI-SADRI AGENTS

Every rr-agent can be thought of as an abductive logic program, equipped with an initial goal. The abducibles are *actions* to be executed as well as *observations* to be performed. Updates, observations and queries are treated uniformly as goals. The abductive logic program can be a temporal theory. For example, the event calculus [72] can be written as an abductive logic program:

$$\begin{aligned} \mathit{holds\_at}(p, t_2) &\leftarrow \mathit{happens}(e, t_1) \wedge (t_1 < t_2) \wedge \mathit{initiates}(e, p) \wedge \\ &\quad\ \mathit{not}\, \mathit{broken}(t_1, p, t_2) \\ \mathit{broken}(t_1, p, t_2) &\leftarrow \mathit{happens}(e, t) \wedge \mathit{terminates}(e, p) \wedge (t_1 < t < t_2). \end{aligned}$$

The first clause expresses that a property p holds at some time t_2 if it is initiated by an event e at some earlier time t_1 and is not broken (i.e., persists) from t_1 to t_2. The second clause expresses that a property p is broken (i.e., does not persist) from a time t_1 to a later time t_2 if an event e that terminates p happens at a time t between t_1 and t_2.

The predicate *happens* is abducible, and can be used to represent both observations, as events that have taken place in the past, or events scheduled to take place in the future. An integrity constraint

$$I_1 \qquad \mathit{happens}(e, t) \wedge \mathit{preconditions}(e, t, p) \wedge \mathit{not}\, \mathit{holds\_at}(p, t) \Rightarrow \mathit{false}$$

expresses that an event e cannot happen at a time t if the preconditions p of e do not hold at time t.

The predicates *preconditions*, *initiates* and *terminates* have application specific definitions, e.g.,

$$\begin{aligned} &\mathit{preconditions}(\mathit{carry\_umbrella}, t, p) \leftarrow p = \mathit{own\_umbrella} \\ &\mathit{preconditions}(\mathit{carry\_umbrella}, t, p) \leftarrow p = \mathit{borrowed\_umbrella} \\ &\mathit{initiates}(\mathit{rain}, \mathit{raining}) \\ &\mathit{terminates}(\mathit{sun}, \mathit{raining}). \end{aligned}$$

Additional integrity constraints might be given to represent reactive behaviour of intelligent agents, e.g.,

$$I_2 \qquad \mathit{holds\_at}(\mathit{raining}, t) \Rightarrow \mathit{happens}(\mathit{carry\_umbrella}, t + 1)$$

or to prevent concurrent execution of actions (events)

$$\mathit{happens}(e_1, t) \wedge \mathit{happens}(e_2, t) \Rightarrow e_1 = e_2.$$

The basic "engine" of a rr-agent is the IFF proof procedure, executed via the following cycle (that is an extension of the cycle of rational agents of Section 10.5):

```
To cycle at time $t$,
    $(i)$ observe any input at time $t$,
   $(ii)$ record any such input,
  $(iii)$ resume the IFF procedure by propagating the inputs,
   $(iv)$ continue applying the IFF procedure, using for steps $(iii)$ and
          $(iv)$ a total of $r$ units of time,
    $(v)$ select an atomic action which can be executed at time $t + r + 2$,
   $(vi)$ execute the selected action at time $t + r + 2$ and record the result,
  $(vii)$ cycle at time $t + r + 3$.
```

The cycle starts at time t by observing and recording any inputs from the environment (steps (i) and (ii)). Steps (i) and (ii) are assumed to take one unit of time each. Then, the IFF proof procedure is applied for r units of time (steps (iii) and (iv)). The amount of resources r available in steps (iii) and (iv) is bounded by some predefined amount n. By decreasing n the agent is more *reactive*, by increasing n the agent is more *rational.* Propagation is applied first (to the new inputs, step (iii)), in order to allow for an appropriate reaction to the inputs. Afterwards, an action is selected and executed, taking care of recording the result (steps (v) and (vi)). Steps (v) and (vi) conjoined are assumed to take one unit of time. Observations can be thought of as inputs from the environment and selected actions as outputs into the environment. Recording an observation as well as the result of an action is achieved by conjoining it to the goals of the agent, as an additional input. From every agent's viewpoint, the environment contains all other agents.

Selected actions correspond to abducible atoms in an answer extracted from a disjunct in a derived goal in a derivation (i.e., an action to be executed is selected from D). The disjunct represents an *intention*, i.e., a (possibly partial) plan executed in stages. A sensible action selection strategy may select actions from the same disjunct (intention) at different iterations of the cycle. Failure of a selected plan is obtained via logical simplification, after having propagated *false* into the selected disjunct.

Actions that are generated in an intention may have times associated with them. The times may be absolute, for example $happens(ring\_bell, 3)$, or may be within a constrained range, for example $happens(leave, t) \wedge (1 < t < 10)$. In step (v), the selected action will either have an absolute time equal to $t + r + 2$ or a time range compatible with an execution time at $t + r + 2$. In the latter case, recording of the result of the execution instantiates the time associated to the action.

Integrity constraints provide a mechanism not only for constraining explanations and plans (for instance, as in I_1), but also for allowing reactive, condition-action type of behaviour (for instance, as in I_2).

11.5 ADDING INTROSPECTION TO RR-AGENTS

In this section, as in the rr-agents' framework, agents are represented as abductive logic programs rather than labelled definite programs as in RCL. Every abductive logic program, in its completed form, is executed within a rr-agents' cycle.

In order to accommodate beliefs of different agents within agents, we give *solve* an additional argument rather than using theory symbols as in RCL (see Section 8.1). The atom $solve(\omega^1, X)$ stands for "Agent ω believes X". We will assume that *solve* can only take names of atoms as second argument. By convention, $solve(X)$ will be an abbreviation for $solve(\omega^1, X)$ within the program P of ω itself.

As in RCL, we incorporate the reflection principle linking base level and metaevaluation level via (a finite set of) clauses to be added to the given (abductive) logic program.

Let $\langle P, \mathcal{A}, I\rangle$ be an abductive logic program and C a clause $H \leftarrow B$ in P. Then, we define a reflection principle $\mathcal{I}$ as:

$$\mathcal{I}(C) = \begin{cases} \mathcal{U}(C) & \text{if } C \text{ is a base level clause,} \\ \mathcal{O}(C) & \text{if } C \text{ is a metaevaluation clause.} \end{cases}$$

$\mathcal{U}$ is defined as in Section 7.2, i.e.,

- If H is $p(t_1, \ldots, t_n)$, then

$$\mathcal{U}(C) = \{ solve([p^1, X_1, \ldots, X_n]) \leftarrow X_1 = \uparrow t_1 \wedge \ldots \wedge X_n = \uparrow t_n \wedge B \ \}.$$

Note that necessarily $p \notin \mathcal{A}$, since we assume that abducible predicates are not defined in P. The reflection principle $\mathcal{O}$ is a modification of the reflection principle $\mathcal{D}$ (Section 7.2) to take into consideration abducible predicates.

- If H is $solve([p^1, t_1, \ldots, t_n])$ and $p \notin \mathcal{A}$, then

$$\mathcal{O}(C) = \{p(x_1, \ldots, x_n) \leftarrow x_1 = \downarrow t_1 \wedge \ldots \wedge x_n = \downarrow t_n \wedge B\}\,.$$

- If H is $solve([X, t_1, \ldots, t_n])$, then

$$\mathcal{O}(C) = \left\{ \begin{array}{ll|l} p(x_1, \ldots, x_n) \leftarrow & x_1 = \downarrow t_1 \wedge & \text{for every } n\text{-ary} \\ & \ldots & \text{predicate symbol} \\ & x_n = \downarrow t_n \wedge & p \notin \mathcal{A} \text{ and } p \neq solve \\ & X = p^1 \wedge B & \end{array} \right\}.$$

- If H is $solve(X)$, then

$$\mathcal{O}(C) = \left\{ \begin{array}{ll|l} p(y_1, \ldots, y_n) \leftarrow & y_1 = \downarrow X_1 \wedge & \\ & \ldots & \text{for every } n\text{-ary} \\ & y_n = \downarrow X_n \wedge & \text{predicate symbol} \\ & X = [p^1, X_1, \ldots, X_n] & p \notin \mathcal{A} \text{ and } p \neq solve \\ & \wedge B & \end{array} \right\}.$$

$\mathcal{I}(P)$ is given by the union of $\mathcal{I}(C)$, for all $C \in P$. In order to have a metalevel representation of provability of abducible predicates, we introduce the notion of abducibility sets.

▶ Let $\langle P, \mathcal{A}, I \rangle$ be an abductive logic program. The **abducibility set** of $\mathcal{A}$, written as , $(\mathcal{A})$, is the set of clauses:

$$, (\mathcal{A}) = \left\{ \begin{array}{ll|l} solve([a^1, X_1, \ldots, X_n]) \leftarrow & X_1 = \uparrow y_1 \wedge & \text{for every } n\text{-ary} \\ & \ldots & \text{predicate sym-} \\ & X_n = \uparrow y_n \wedge & \text{bol } a \in \mathcal{A} \\ & a(y_1, \ldots, y_n) & \end{array} \right\}.$$

The intended connection among base level, metaevaluation level and abducible atoms is captured by the following definition.

▶ Let $\langle P, \mathcal{A}, I \rangle$ be an abductive logic program.

The **associated program** of P with respect to $\mathcal{A}$, written as $\Delta(P, \mathcal{A})$, is defined as:

$$\Delta(P, \mathcal{A}) = P \cup \mathcal{I}(P) \cup , (\mathcal{A}).$$

The **associated integrity constraints** of I with respect to $\mathcal{A}$, written as $\Delta(I, \mathcal{A})$, is defined as:

$$\Delta(I, \mathcal{A}) = I \cup \{solve([a^1, X_1, \ldots, X_n]) \wedge x_1 = \downarrow X_1 \wedge \ldots \wedge x_n = \downarrow X_n \\ \Rightarrow a(x_1, \ldots, x_n)\},$$

for every n-ary predicate symbol $a \in \mathcal{A}$.

The **meta-abductive logic program** associated with $\langle P, \mathcal{A}, I\rangle$ is:

$$\langle \Delta(P, \mathcal{A}),\ \mathcal{A},\ \Delta(I, \mathcal{A})\rangle.$$

The addition of the new integrity constraints in $\Delta(I, \mathcal{A})$ allows the agent to propagate (and thus compute the consequences of) any new information it receives about abducible predicates in whatever (metaevaluation level or base level) form, without any need to alter the original set of integrity constraints, I.

Remark. In general it is not possible to extend a logic program directly with the axiom schemas

$$A \leftarrow solve(A^1) \quad \text{and} \quad solve(A^1) \leftarrow A,$$

because the completion of the extended logic program may contain circular definitions. This is the case, for instance, of the completion of the program P that extends the program

$$\{a \leftarrow b\}$$

with the axiom schemas above.

$$P = \left\{ \begin{array}{lll} a \leftarrow b & a \leftarrow solve(a^1) & b \leftarrow solve(b^1) \\ & solve(a^1) \leftarrow a & solve(b^1) \leftarrow b \end{array} \right\}$$

$$comp(P) = \left\{ \begin{array}{l} a \leftrightarrow b \vee solve(a^1) \\ b \leftrightarrow solve(b^1) \\ solve(X) \leftrightarrow [X = a^1 \wedge a] \vee [X = b^1 \wedge b] \end{array} \right\}$$

□

11.6 ADDING COMMUNICATION TO RR-AGENTS

In this section, we interpret *tell*(X, Y) and *told*(X, Y) as abducible predicates in meta-abductive logic programs. As for *solve* we can give *tell* and *told* an additional argument instead of introducing labels, to represent communication between agents. For simplicity, we will abbreviate *tell*(ω^1, X, Y) (resp., *told*) within the program P of agent ω itself as *tell*(X, Y).

Example 11.5 Let an agent ω be represented by the abductive logic program $\langle P, \mathcal{A}, I\rangle$ with:

$$P = \left\{ \begin{array}{l} \mathit{solve}(X) \leftarrow \mathit{told}(A, X) \\ \mathit{desire}(y) \leftarrow y = \mathit{car} \\ \mathit{good\_price}(p, x) \leftarrow p = \mathit{0} \end{array} \right\}$$

$$\mathcal{A} = \{ \ \mathit{tell}, \mathit{told}, \mathit{offer} \ \}$$

$$I = \left\{ \begin{array}{r} \mathit{desire}(x) \wedge \mathit{told}(A, [\mathit{good\_price}^1, P, X]) \wedge X = \uparrow x \\ \Rightarrow \mathit{tell}(A, [\mathit{offer}^1, P, X]) \end{array} \right\}.$$

Namely, ω believes anything it is told (by any other agent) and it desires to have a car. The third clause in P says that anything that is free is at a good price. Moreover, if the agent desires something and it is told (by some other agent) of a good price for it, then it makes an offer to the other agent, by telling it. Note that a more accurate representation of the integrity constraint should include time.

The corresponding meta-abductive logic program $\langle \Delta(P, \mathcal{A}), \mathcal{A}, \Delta(I, \mathcal{A})\rangle$ is:

$\Delta(P, \mathcal{A}) = P \cup \mathcal{I}(P) \cup \Gamma(\mathcal{A})$, where

$\mathcal{I}(P)$ is:

$$\left\{ \begin{array}{ll} \mathit{solve}([\mathit{desire}^1, X_1]) \leftarrow X_1 = \uparrow y \wedge y = \mathit{car} & \\ \mathit{solve}([\mathit{good\_price}^1, X_1, X_2]) \leftarrow X_1 = \uparrow p \wedge X_2 = \uparrow x \wedge p = \mathit{0} & \\ \mathit{desire}(x_1) \leftarrow x_1 = \downarrow X_1 \wedge X = [\mathit{desire}^1, X_1] \wedge \mathit{told}(A, X) & \\ \mathit{good\_price}(x_1, x_2) \leftarrow & x_1 = \downarrow X_1 \wedge x_2 = \downarrow X_2 \wedge \\ & X = [\mathit{good\_price}^1, X_1, X_2] \wedge \mathit{told}(A, X) \end{array} \right\}$$

$\Gamma(\mathcal{A})$ is:

$$\left\{ \begin{array}{l} \mathit{solve}([\mathit{tell}^1, X_1, X_2]) \leftarrow X_1 = \uparrow A \wedge X_2 = \uparrow X \wedge \mathit{tell}(A, X) \\ \mathit{solve}([\mathit{told}^1, X_1, X_2]) \leftarrow X_1 = \uparrow A \wedge X_2 = \uparrow X \wedge \mathit{told}(A, X) \\ \mathit{solve}([\mathit{offer}^1, X_1, X_2]) \leftarrow X_1 = \uparrow p \wedge X_2 = \uparrow x \wedge \mathit{offer}(p, x) \end{array} \right\}$$

and $\Delta(I, \mathcal{A})$ is:

$$I \cup \left\{ \begin{array}{l} \mathit{solve}([\mathit{tell}^1, X_1, X_2]) \wedge A = \downarrow X_1 \wedge X = \downarrow X_2 \Rightarrow \mathit{tell}(A, X) \\ \mathit{solve}([\mathit{told}^1, X_1, X_2]) \wedge A = \downarrow X_1 \wedge X = \downarrow X_2 \Rightarrow \mathit{told}(A, X) \\ \mathit{solve}([\mathit{offer}^1, X_1, X_2]) \wedge p = \downarrow X_1 \wedge x = \downarrow X_2 \Rightarrow \mathit{offer}(p, x) \end{array} \right\}.$$

$\square$

Given an abductive logic program $\langle P, \mathcal{A}, I\rangle$, corresponding to some agent, and the associated meta-abductive logic program $\langle \Delta(P, \mathcal{A}), \mathcal{A}, \Delta(I, \mathcal{A})\rangle$, the agent's cycle applies the IFF procedure with

$$T = comp_{\overline{\mathcal{A}}}(\Delta(P, \mathcal{A})).$$

Example 11.6 Let $\langle P, \mathcal{A}, I\rangle$ be the abductive logic program of Example 11.5. Assume that ω has the input observation (step (i) of the cycle):

$$told(\phi^1, [good\_price^1, 50^1, car^1]),$$

meaning that ω has been told by an agent ϕ of a good price (of 50) for a car. The initial goal G (step (ii) of the cycle) of ω is:

$$told(\phi^1, [good\_price^1, 50^1, car^1]).$$

Then, the IFF proof procedure is applied (steps (iii) and (iv) of the cycle) to $G \wedge \Delta(I, \mathcal{A})$. A computed answer for G is:

$$\begin{array}{lcl} D & = & \{told(\phi^1, [good\_price^1, 50^1, car^1]), tell(\phi^1, [offer^1, 50^1, car^1])\} \\ \sigma & = & \{\}. \end{array}$$

Finally, the agent ω selects an action from D and executes it (steps (v) and (vi)). In this case, the selected action is $tell(\phi^1, [offer^1, 50^1, car^1])$, meaning that ω proactively makes an offer to ϕ by telling it. $\square$

The rr-agent architecture has the following properties:

- rr-agents are known by their symbolic names,
- when a rr-agent sends a message, it directs that message to a specific addressee,
- when a rr-agent receives a message, it knows the sender of that message,
- messages may get lost.

Both *tell* and *told* are treated as actions: everytime (the cycle of) an agent ω selects a communicative action, i.e., an action of the form

$$told(\phi^1, X) \qquad \text{or} \qquad tell(\phi^1, X),$$

ω will attempt to execute it. If the attempt is successful, the record

$$told(\phi^1, X) \qquad \text{or} \qquad tell(\phi^1, X)$$

is conjoined to the goals of agent ω, as an additional input. If the attempt is not successful, the record

$$told(\phi^1, X) \Rightarrow false \qquad \text{or} \qquad tell(\phi^1, X) \Rightarrow false$$

is conjoined to the goals of agent ω, as an additional input. In the next section we will formalise an example showing how proactive communication is achieved by executing the proof procedure within cycle.

11.7 EXAMPLE

The following example demonstrates the running of the IFF proof procedure within cycle, action selection, proactive communication whereby one agent volunteers information to another, and how during the planning phase such information can help in the choice of intention.

The example is as follows: an agent wishes to register for a conference, let us say Jelia, that is to take place in Paris on the 10th and to make travel arrangements to go to Paris on the 10th (for simplicity we omit the month). So the agent's original goal is the conjunction:

$$register(Jelia) \wedge travel(Paris, 10).$$

In this section, to enhance readability we write variables as words with more than one character and we distinguish them by underling their first character. Thus, for instance, $\underline{c}ity$ is a variable and $\underline{C}ity$ (the word starts with capital letter) is a metavariable. The agent has the following program, P, integrity constraints, I, and abducibles $\mathcal{A}$.

P is:

C_1 $\quad travel(\underline{c}ity, \underline{d}ate) \leftarrow go(\underline{c}ity, \underline{d}ate, train)$

C_2 $\quad travel(\underline{c}ity, \underline{d}ate) \leftarrow go(\underline{c}ity, \underline{d}ate, plane)$

C_3 $\quad early\_registration(Jelia) \leftarrow send\_form(Jelia, \underline{d}ate) \wedge (\underline{d}ate < 3)$

C_4 $\quad go(\underline{c}ity, \underline{d}ate, \underline{m}eans) \leftarrow book(\underline{c}ity, \underline{d}ate, \underline{m}eans)$

C_5 $\quad book(\underline{c}ity, \underline{d}ate, \underline{m}eans) \leftarrow$

$\qquad \underline{c}ity = \downarrow\underline{C}ity \wedge \underline{d}ate = \downarrow\underline{D}ate \wedge \underline{m}eans = \downarrow\underline{M}eans \wedge$

$\qquad told(ticket\_agent^1, [available^1, \underline{C}ity, \underline{D}ate, \underline{M}eans]) \wedge$

$\qquad tell(ticket\_agent^1, [reserve^1, \underline{C}ity, \underline{D}ate, \underline{M}eans])$

C_6 $\quad solve(\underline{X}) \leftarrow told(ticket\_agent^1, \underline{X})$

Clauses C_1 and C_2 say that one travels to a $\underline{c}ity$ on a given $\underline{d}ate$ if one goes there on that $\underline{d}ate$ by train or by plane. C_3 says that the deadline for early registration for Jelia is the 3rd. C_4 says that one goes to a $\underline{c}ity$ on a given $\underline{d}ate$ by some $\underline{m}eans$ if one makes a booking for that journey. C_5 says that one makes a booking if the ticket_agent confirms availability of ticket and one makes a reservation. Note that in a more thorough representation we would represent the transaction time of when a booking is made (which must be before the time of travel). In that case we will have an extra argument in *tell* and *told* that represent such transaction times. We will ignore this issue here for simplicity.

I is:

$$I_1 \qquad register(\underline{c}onference) \Rightarrow early\_registration(\underline{c}onference)$$

$$I_2 \qquad go(\underline{c}ity, \underline{d}ate, \underline{m}eans) \wedge strike(\underline{c}ity, \underline{d}ate, \underline{m}eans) \Rightarrow false$$

I_1 states the agent's departmental policy that anyone registering at a conference must take advantage of early registration. I_2 states that one cannot use a means of transportation which is subject to a strike.

$\mathcal{A}$ is:

$$\{send\_form, tell, told, register, available, reserve, strike\}.$$

Now suppose that the cycle of the agent starts at time 1 and that the agent does not observe any input. Thus, the cycle effectively starts at step (iv) by applying the IFF proof procedure (using $\langle \Delta(P,\mathcal{A}), \mathcal{A}, \Delta(I,\mathcal{A})\rangle$ which we do not show here) to the goal obtained by conjoining the original goal and the integrity constraints $\Delta(I,\mathcal{A})$. By repeatedly applying the IFF proof procedure, the goal is transformed (within the initial cycle or within some later iteration, depending on the resource parameter r) into

$$register(Jelia) \wedge travel(Paris, 10) \wedge send\_form(Jelia, \underline{d}ate) \wedge (\underline{d}ate < 3)$$

At this point cycle can select the action $send\_form(Jelia, \underline{d}ate)$ to perform if time is less than 3. If the agent has been too slow and time 3 has already passed, the agent has failed its goals. Suppose the agent succeeds. Note that at any time any other agent can send this agent a message. So suppose the ticket agent sends a message that trains are on strike in Paris on the 10th, i.e., at some iteration of cycle at step (i) the agent receives the input

$$told(ticket\_agent^1, [strike^1, Paris^1, 10^1, train^1]).$$

In that iteration of cycle this information is propagated (step (iii)) and the simplified constraint

$$go(Paris, 10, train) \Rightarrow false$$

is added to the goal. Meanwhile the sub-goal $travel(Paris, 10)$ is unfolded (step (iv)) into

$$go(Paris, 10, train) \vee go(Paris, 10, plane).$$

The information about the strike will be used to remove the first possibility (i.e., the first disjunct) leaving only

$$go(Paris, 10, plane)$$

which will become the agent's intention. By applying few steps of unfolding and equality rewriting, we obtain the plan:

$$told(ticket\_agent^1, [available^1, Paris^1, 10^1, plane^1]) \wedge$$
$$tell(ticket\_agent^1, [reserve^1, Paris^1, 10^1, plane^1]).$$

The appropriate actions will be selected during an iteration of cycle.

11.8 FORMAL PROPERTIES

In this section, we informally argue that the results of soundness and completeness of the IFF proof procedure still hold for meta-abductive logic programs. In the original definition of the IFF proof procedure equality rewriting is based on a set of rules (presented in Example 11.3) that are sound for the Clark's equality theory, *CET*. In our framework instead, in order to take into account names of the metalanguage, we have assumed that, given an equality theory E that extends *CET* to formalise names, the equality rewriting rule simulates the corresponding E-unification algorithm and the application of substitutions. $E = CET \cup E'$ has to be such that E' is sufficiently complete and consistent (see the remark on p. 116). The two conjectures presented in this section require that equality rewriting is sound for E, i.e., if a formula F_2 is obtained by a formula F_1 by equality rewriting, then we have that $E \models_3 F_1 \leftrightarrow F_2$, where $\leftrightarrow$ denotes "if and only if". We implicitly make this assumption.

We use the following definitions.

- ▶ Let $\langle P, \mathcal{A}, I \rangle$ be an abductive logic program and G an initial goal. Let D be a finite set of ground abducible atoms and σ a substitution such that $G\sigma$ is ground. Then, (D, σ) is an **answer** to G.

Let $\models_3$ be the logical consequence in Kunen's three-valued logic [73].

▶ Let E be an equality theory and $\langle P, \mathcal{A}, I\rangle$ an abductive logic program. Let G be an initial goal and (D, σ) an answer to G. Then, (D, σ) is a **correct answer** to G if and only if

1. $comp(\Delta(P, \mathcal{A}) \cup D) \cup E \models_3 G\sigma$ and
2. $comp(\Delta(P, \mathcal{A}) \cup D) \cup E \models_3 \Delta(I, \mathcal{A})$.

By noting that meta-abductive logic programs are abductive logic programs, the proofs of the following two conjectures are based on the results of soundness and completeness of the IFF proof procedure [52].

Conjecture 11.7 (*Soundness*) *Let $\langle P, \mathcal{A}, I\rangle$ be an abductive logic program and G an initial goal.*

(*i*) *If there exists a successful derivation of the form $F_1 = G \wedge \Delta(I, \mathcal{A}), \ldots, F_n = N \vee Rest$ such that (D, σ) is an answer extracted from N, then (D, σ) is a correct answer to G.*

(*ii*) *If there exists a derivation $F_1 = G \wedge \Delta(I, \mathcal{A}), \ldots, F_n = false$, then we have $comp_{\overline{\mathcal{A}}}(\Delta(P, \mathcal{A})) \cup E \cup \Delta(I, \mathcal{A}) \models_3$ not G.*

Conjecture 11.8 (*Completeness*) *Let $\langle P, \mathcal{A}, I\rangle$ be an abductive logic program and G an initial goal. If (D', σ) is a correct answer to G, then there exists a successful derivation $F_1 = G \wedge \Delta(I, \mathcal{A}), \ldots, F_n = N \vee Rest$ such that (D, σ) is an answer extracted from N and $D \subseteq D'$.*

11.9 COMMUNICATION MODELLING

Typically, communication processes are understood by appealing to speech act theory [102]. In speech act theory a message can be identified with an *illocution.* An illocution has two parts: an *illocutionary type* and a *proposition.* The illocutionary types include assertives, directives, commissives and permissives. For instance, the speaker may be asserting a fact or requesting a service. The proposition describes the state of the world that is, respectively, asserted or requested. Several types of primitive messages used in agents communication languages can be described by using the above basic illocutionary types (see, e.g., [119]).

In this section we show how the basic illocutionary types can be described within the icrr-agent's framework.

Assertives

Assert is used to inform another agent about a fact. Then, it is up to the receiving agent to determine whether or not to believe it.

The effects of an assertion can be described as:

$$\begin{aligned} know(\omega^1, [believe^1, \phi^1, X]) &\leftarrow told(\phi^1, [assert^1, \phi^1, \omega^1, X]) \\ believe(\omega^1, X) &\leftarrow told(\phi^1, [assert^1, \phi^1, \omega^1, X]) \wedge \\ &\qquad authority(\phi^1, [assert^1, \phi^1, \omega^1, X]). \end{aligned}$$

If an agent ϕ asserts a fact X to an agent ω, then ω knows that ϕ believes X, and in case ϕ has authority to assert it, then ω also believes X.

Clauses and integrity constraints can be asserted. Let *if* and *imply* name $\leftarrow$ and $\Rightarrow$, respectively.

$$\begin{aligned} &solve(X) \leftarrow told(\phi^1, X \; if \; Y) \wedge solve(Y) \\ &told(\phi^1, X \; imply \; Y) \wedge solve(X) \Rightarrow solve(Y) \end{aligned}$$

Permissives

Declare is used to create new facts.

$$solve(Y) \leftarrow Y = \downarrow X, told(\phi^1, [declare^1, X]) \wedge authority(\phi^1, [declare^1, X])$$

A fact X holds in the theory of an agent ω if X has been declared by an authorized agent ϕ. This kind of message, for example, can be used to explicitly give authorizations to other agents, e.g., I give you permission to access the employee database. Thus, if ω has the input observation:

$$told(\phi^1, [declare^1, [authority^2, \omega^3, [access^3, employee\_database^3]]])$$

and its underlying theory contains the constraint:

$$authority(\omega^1, X) \Rightarrow solve(X),$$

then ω can access the employee database.

Directives

Direct is used to give orders or requests to other agents.

(i) Request

$$\begin{aligned} &told(\phi^1, [ask^1, X]) \wedge authority(\phi^1, [ask^1, X]) \wedge solve(X) \Rightarrow tell(\phi^1, X) \\ &told(\phi^1, [ask^1, X]) \wedge authority(\phi^1, [ask^1, X]) \wedge not \; solve(X) \Rightarrow tell(\phi^1, not^1 \; X) \end{aligned}$$

Assume the above clauses belong to a theory underlying an agent ω. The first clause means that if ω has the input observation $told(\phi^1, [ask^1, X])$, meaning that ω has been asked X by an agent ϕ, ϕ has authority to ask it and ω can prove X, then ω tell ϕ that X.

(ii) Order

The effect of a directive depends on the authorization of the speaker. Suppose that the underlying theory of an agent ω contains:

$$\begin{aligned} solve([direct^1, \phi^1, \omega^1, X]) \leftarrow\ & told(\phi^1, [direct^1, \phi^1, \omega^1, X]) \wedge \\ & authority(\phi^1, [direct^1, \phi^1, \omega^1, X]) \end{aligned}$$

$$obligation(\omega^1, \phi^1, X) \leftarrow direct(\phi^1, \omega^1, X)$$

$$obligation(\omega^1, \phi^1, X) \Rightarrow solve(X).$$

The second clause says that if an agent ϕ orders an agent ω to perform X, then ω has the obligation towards ϕ to perform X, and the integrity constraint makes this obligation effective by making ω to execute X. For example, if ω has the input observation $told(\phi^1, [direct^1, \phi^1, \omega^1, prepare\_coffee^1])$, then ω has the obligation to *prepare_coffee* towards agent ϕ.

If agent ϕ is not authorized the only effect of the order is that agent ω knows that agent ϕ intends him to perform X:

$$know(\omega^1, [intend^1, \phi^1, \omega^1, X]) \leftarrow told(\phi^1, [direct^1, \phi^1, \omega^1, X]).$$

Commissives

Commit is used to create obligations for oneself. If a request is made by another agent which has no authorization, then the agent can honour the request by committing itself to the action.

(i) Obligation

$$obligation(\omega^1, \phi^1, X) \leftarrow commit(\omega^1, \phi^1, X) \wedge told(\phi^1, [declare^1, \omega^1, X]))$$

$$obligation(\omega^1, \phi^1, X) \Rightarrow solve(X)$$

If an agent ω commits itself towards an agent ϕ to perform X and agent ϕ declared that ω is permitted to perform X, then ω has the obligation towards ϕ to perform X.

(ii) Prohibition
An agent ω does not honour an obligation if some *conditions* are met,

$$solve(X) \wedge conditions \Rightarrow false.$$

In summary, we have presented an approach to agents that can reason about their own beliefs as well as beliefs of other agents and that can communicate with each other. The approach results from the combination of the approach to agents in [71] and the approach to meta-reasoning and communication in [15, 34]. We have illustrated the approach by means of a number of examples.

The approach needs to be extended in a number of ways. For simplicity, we have ignored the treatment of time. However, time plays an important role in most agent applications and should be explicitly taken into account.

We have considered only two communication performatives, *tell* and *told*. Existing communication languages, e.g., [32], consider additional performatives, e.g., *deny*, *achieve* and *unachieve*. We are currently investigating whether some of these additional performatives could be defined via communication protocols, as definitions and integrity constraints within our framework.

The primitive *told* is used to express both *active* request for information and *passive* communication. The two roles should be separated out, possibly with the addition of a third predicate *ask*, distinguished from *told*. Then the predicate *told* could be defined in terms of *ask* and *tell*, rather than be an abducible. For example, an agent ω is told of X by another agent ϕ if and only if ϕ tells ω of X or ω actively asks ϕ about X and ϕ gives a positive answer.

We have implicitly assumed that different agents share the same content language. However, this assumption is not essential. Indeed, “translator agents” could be defined, acting as mediators between agents with different content languages. This is possible by virtue of the metalogic features of the language.

Conclusion

We summurise our contributions and examine previous work in the literature and possible relationships to ours.

12.1 THESIS CONTRIBUTION

The thesis has introduced the concept of reflection principles 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*, has been 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.

RCL allows its users to specify a variety of deductive systems, given through axioms and inference rules. The syntax of the language of the deductive systems that can be specified in *RCL* is based on a language, HC^+, that augments the language of Horn clauses to contain names of the expressions of the language itself. This makes it possible to specify deductive systems that are able to represent knowledge and metaknowledge about a problem domain and to perform metareasoning. *RCL* has been defined so as to leave significant freedom in the representation of names, allowing users to choose the most appropriate encoding for the application domain at hand.

In *RCL* encodings are formalised as equality theories and are characterized computationally by rewrite systems. We have presented an encoding, *NT*, for compositional names of HC^+, and a corresponding rewrite system, *NR*. We have shown that *NR* correctly characterizes *NT* in terms of the notion of "adequateness". Then, we have shown how *NR* can be embedded into the unification algorithm, *UN*, of HC^+ and we have investigated which properties *UN* has to satisfy when integrated into a computational framework. In particular, we have proved that *UN* is terminating and sound for *NT*.

RCL provides its users with the possibility to express an inference rule R of a deductive system *DS* in the form of a reflection principle $\mathcal{R}$. In particular, the user can express R as a function $\mathcal{R}$ from clauses (which constitute the antecedent of R) to sets of clauses (which constitute the consequent of R). The novelty of the approach is that R, once defined by $\mathcal{R}$, is immediately executable. Being the antecedent of the inference rule a single Horn clause seems really not to be a limitation, or at least for the application domains considered. In fact, if we need several Horn clauses as premises, we can encode their information into one Horn clause.

The model-theoretic and fixed point semantics of *DS* are obtained as a side effect of the specification. In fact, after having formalised the encoding of *DS* as an equality theory E, having expressed the axioms of *DS* as a definite program P in the metalanguage HC^+, and having defined the inference rules of *DS* as a reflection principle $\mathcal{R}$, then the declarative semantics of *DS* is provided in terms of the least reflective E-model $M^{\mathcal{R}}_{(P,E)}$ of the logic program (P, E). $M^{\mathcal{R}}_{(P,E)}$ has also been characterized as the least fixed point of a mapping, called $T^{\mathcal{R}}_{(P,E)}$, defined over E-interpretations of (P, E), that extends the usual $T_{(P,E)}$ to take $\mathcal{R}$ into account.

We have characterized reflection both theoretically and procedurally. From the procedural point of view we have proposed a proof-theoretic extension of SLD-resolution, called SLD$^{\mathcal{R}}$-resolution, based on the concept of state. Such an extension is reminiscent of the procedural behaviour of constraint programming languages, where the computation of values (constraints) is delayed until we have enough information to compute them. This corresponds in our approach to the idea of delaying the computation of names of expressions if they are not ground. In the inference process, reflection principles are not applied at once to the whole logic program, since the resulting program $(P \cup \mathcal{R}(P), E)$ may in general have a large number of clauses. Rather, reflection principles are applied only as necessary, thus computing the reflection axioms "on the fly". In terms of SLD$^{\mathcal{R}}$-resolution, this corresponds to select one clause C from P and then to choose the input clause (of the SLD$^{\mathcal{R}}$-resolution step) from the set $\{C\} \cup \mathcal{R}(C)$. SLD$^{\mathcal{R}}$-resolution has been proved to be sound and complete with respect to the least reflective

E-model of (P, E), provided that the E-unification algorithm, which is a parameter of SLD$^{\mathcal{R}}$-resolution, is sound for E.

We have argued that *RCL* is a practical and powerful computational system and we have provided evidence in form of examples from three different application domains.

Finally, we have shown how our approach to reflection is powerful and flexible enough to be integrated into different frameworks. In particular, we have embedded the reflection principles proposed to model introspective, communicative agents (ic-agents), into the framework of rational, reactive agents (rr-agents) proposed by Kowalski and Sadri. The resulting kind of agents combine the characteristics of both.

12.2 RELATED WORK AND CONCLUDING REMARKS

In the introduction we gave general references to the ample subject of meta-level architectures and reflection. In this section, we make an attempt to more specifically relate our approach to other proposals advanced in several contexts, since we wish to emphasise 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.

1. 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 & Sergot [74] suggest “to write a logic program implicitly by stating the defining property which characterizes 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 [10] 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 metaprograms.

   Fuchs [51] observes that “since the beginning of logic programming it has been recognised 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 transformation strategies. Transformations generate equivalent programs in that the least Herbrand model and the computed answers are preserved.

Yokomori [124] 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 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 Chapter 7.

2. In the automated theorem proving field, several authors have raised the issue of supplementing the prover with additional information, either to improve its performance or to enhance its proving abilities.

   Kerber [68] shows the drastic performance improvements that can be achieved by the "incorporation of declarative knowledge into an automated theorem prover that can be utilised in the search for a proof". Following Polya [94] and Bundy [25], Kerber considers analogy as the knowledge useful for empowering the reasoning, in a way that "does not enlarge the possibilities of deduction in principle, but is helpful in guiding the search for a proof". He advocates the need to develop an epistemologically appropriate language, recognising that it is important to find the adequate level of abstraction, though neither a formalisation of analogy nor a line of development of such a language is addressed in that paper.

   Pastre [88] discusses at length the necessity of "metaknowledge (general and mathematical), structured and used in the same manner as knowledge". Interestingly enough, the paper suggests the use of non-rigorous (unsound) proof methods as a means of creating intermediate objects. The proposed system MUSCADET "is slower than some other systems for certain theorems, the easier ones, but, contrary to the previous systems, it is also able to work in difficult fields".

   The *Nuprl* theorem prover [5, 33] is a goal-driven, extensible, tactic-oriented prover, in which the user can safely add new tactics and decision procedures by writing them directly in the logic, which has constructive type theory as its underlying theory and functional programming language as its computational system. Reflection is used to raise the level of abstraction of proofs and this increases their efficiency and explianability. Tactic rules and reflective rules, however, can be eliminated from proofs, so there is no increase in the axiomatic basis of the theory (it was a design decision for the extension to be conservative).

In the automated theorem proving field, it is common to make recourse to either metatheoretic or higher-order features as, for instance, by Stabler [108] for the former and by Boy de la Tour & Caferra [20] for the latter.

Pfenning [93] calls "logical frameworks" a meta-language for the specification of deductive 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 methods of the intended application domain". While surveying several frameworks, he remarks that "research in logical framework is still in its infancy".

3. In the fields of knowledge representation and machine learning, several authors have expressed the view that metalevel inference and reflection can be used to represent multiple sources of knowledge (theories or agents) or/and multiple forms of reasoning and learning. For the sake of this discussion, we mention here just three of them.

   Lownie [80] defines a non-clausal full first-order language with multiple theories: "sets of statements about more than one domain, or for different aspects of the same domain organised into separate contexts." The reflective architectures consisting of towers of meta-theories "provide a well-motivated framework for integrating specialised control with general reasoning in knowledge-based systems", and Smith's procedural reflection [106] is used for "establishing a correspondence between a theory and any of its meta-theories."

   Arcos & Plaza [8] view learning by analogy as metalevel inference, where "the meta-theory contains knowledge that allows to deduce how to extend the model of the base theory", and propose an impass-driven architecture based on a frame language with reflection.

   Schroeder-Heister [101] uses reflection for hypothetical reasoning. Our reflection principles are similar to his definitional reflection in that they are viewed as inference schemata rather than as logical axioms.

   We refer the reader to the literature mentioned in the introduction for many other metalevel 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 thesis 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, reasoning 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 Chapters 7, 8 and 9.

Considering in particular the application of the general framework presented in the thesis to the field of metalogic languages, Reflective Prolog (Chapter 7) has been compared to the other main approaches in [38]. A more recent approach, not considered there, is that of [57] which is similar to Reflective Prolog concerning 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 *demo*. *demo* appears explicitly in the body of clauses and has two arguments: the name of a theory and the name of a formula. It is interesting to notice that this approach could be easily modelled in *RCL*: theory communication could be modelled as in Chapter 8, using *demo* on both sides (instead of *tell*/*told*), and *demo* could be forced to convey provable formulae by means of the reflection principle $\mathcal{U}$ (Reflection up) of Chapter 7, with *demo* instead of *solve*.

Finally, let us review how the present thesis 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 [36] and fully defined formally in [38]. The system was then augmented with a reflective, non-monotonic negation apt to represent non-monotonic reasoning [37]. A formalisation of analogical reasoning in this reflective logic was elaborated in [39]. Reflection was used to represent communication among different theories/agents in [34] and [15]. The idea that a common view underlying such diverse contexts could be systematised in the unifying framework of reflection principles was first advanced in [35].

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 unification [13, 14]. The paper [16] first represents a new attempt to both clarify the role of reflection principles at the knowledge level and to formalise it at this enhanced technical level.

We now wish to conclude the thesis 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 (e.g. in [83, 31]). 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 near future, *RCL* will be fully implemented, taking as a starting point the existing implementation of Reflective Prolog, which is fully functional at the Logic Programming Lab of the Computer Science Department of the University of Milano, where it has been used for several applications.

Bibliography

1. H. Abramson and M. H. Rogers, editors. *Meta-Programming in Logic Programming*, Cambridge, Mass., 1989. MIT Press.

2. L. C. Aiello, C. Cecchi, and D. Sartini. Representation and use of metaknowledge. *Proc. of the IEEE*, 74:1304–1321, 1986.

3. L. C. Aiello, D. Nardi, and M. Schaerf. Reasoning about knowledge and reasoning in a meta-level architecture. *Intl. J. of Applied Intelligence*, 1, 1991.

4. L. C. Aiello, D. Nardi, and M. Schaerf. Reasoning about knowledge: the meta-level approach. In *Proc. Scandinavian Conf. on Artificial Intelligence*, pages 4–18, Copenhagen, 1991. IOS Press.

5. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science, Piladelphia (PA), June 4-7 1990*, pages 95–105. IEEE Computer Society Press, 1990.

6. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 493–574. Elsevier, Amsterdam, 1990.

7. K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

8. J. L. Arcos and E. Plaza. A reflective architecture for integrated memory-based learning and reasoning. In S. Wess, K.-D. Althoff, and M. M. Richter, editors, *Topics in Case-Based Reasoning (Proc. EWCBR–93)*, LNAI 837, pages 327–339, Berlin, 1994. Springer-Verlag.

9. G. Attardi and M. Simi. Meta-level reasoning across viewpoints. In T. O'Shea, editor, *Proc. European Conf. on Artificial Intelligence*, pages 315–325, Amsterdam, 1984. North-Holland.

10. D. Barker-Plummer. Cliche programming in Prolog. In M. Bruynooghe, editor, *Proc. Second Workshop on Meta-Programming in Logic*, pages 247–256. Dept. of Comp. Sci., Katholieke Univ. Leuven, 1990.

11. J. Barklund. Metaprogramming in logic. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 33, pages 205–227. M. Dekker, New York, 1995.

12. J. Barklund, K. Boberg, P. Dell'Acqua, and M. Veanes. Meta-programming with theory systems. In K. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 195–226. MIT Press, Cambridge, Mass., 1995.

13. J. Barklund, S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. SLD-resolution with reflection. In M. Bruynooghe, editor, *Logic Programming – Proc. 1994 Intl. Symp.*, pages 554–568, Cambridge, Mass., 1994. MIT Press.

14. J. Barklund, S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Semantical properties of encodings in logic programming. In J. Lloyd, editor, *Logic Programming – Proc. 1995 Intl. Symp.*, pages 288–302, Cambridge, Mass., 1995. MIT Press.

15. J. Barklund, S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Metareasoning agents for query-answering systems. In T. Andreasen, H. Christiansen, and H. Legind Larsen, editors, *Flexible Query-Answering Systems*, pages 103–122. Kluwer Academic Publishers, Boston, Mass., 1997.

16. J. Barklund, S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Reflection Principles in Computational Logic. Submitted to J. of Logic and Computation, 1997.

17. J. Barklund, S. Costantini, and F. van Harmelen, editors. *Proc. Workshop on Meta Programming and Metareasonong in Logic, post-JICSLP96 workshop*, Bonn (Germany), 1996. UPMAIL technical Report No. 127 (Sept. 2, 1996), Computing Science Dept., Uppsala Univ.

18. G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Philos. Soc.*, 31:433–454, 1935.

19. K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-Å. Tärnlund,

editors, *Logic Programming*, pages 153–172. Academic Press, London, 1982.

20. T. Boy de la Tour and R. Caferra. Proof analogy in interactive theorem proving: A method to express and use it via second order pattern matching. pages 95–99. American Association for Artificial Intelligence, 1987.

21. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J. W. Lloyd, editor, *Computational Logic*, pages 117–134. Springer-Verlag, Berlin, 1990.

22. R. A. Brooks. A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation*, 2(1):14–23, 1986.

23. R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

24. M. Bruynooghe, editor. *Proc. Second Workshop on Meta-Programming in Logic*, Leuven (Belgium), 1990. Dept. of Comp. Sci., Katholieke Univ. Leuven.

25. A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

26. R. Carnap. *The Logical Syntax of Language.* Kegan Trench Trubner, London, 1937.

27. I. Cervesato and G. Rossi. Logic meta-programming facilities in 'Log. In A. Pettorossi, editor, *Meta-Programming in Logic*, LNCS 649, pages 148–161, Berlin, 1992. Springer-Verlag.

28. A. Chavez and P. M. Kasbah: An agent marketplace for buying and selling goods. In B. Crabtree and N. Jennings, editors, *Proc. 1st Intl. Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 75–90. The Practical Application Company, 1996.

29. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases.* Plenum Press, New York, 1978.

30. K. L. Clark. Logic-programming schemes and their implementations. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 487–541. MIT Press, Cambridge, Mass., 1991.

31. M. G. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proc. Reflection '96*, pages 263–288. Xerox PARC, 1996.

32. P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In V. Lesser, editor, *Proc. 1st Intl. Conf. on Multiagent Systems*, AAAI Press, pages 65–72. MIT Press, 1995.

33. R. L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation, NATO ASI Series F*, volume 139, pages 109–144, Springer-Verlag, 1995.

34. S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Reflective agents in metalogic programming. In A. Pettorossi, editor, *Meta-Programming in Logic*, LNCS 649, pages 135–147, Berlin, 1992. Springer-Verlag.

35. S. Costantini, P. Dell'Acqua, and G. A. Lanzarone. Extending Horn clause theories by reflection principles. In Craig MacNish, David Pearce, and Luis Moniz Pereira, editors, *Logics in Artificial Intelligence*, LNAI 838, Berlin, 1994. Springer-Verlag.

36. S. Costantini and G. A. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, *Proc. 6th Intl. Conf. on Logic Programming*, pages 218–233, Cambridge, Mass., 1989. MIT Press.

37. S. Costantini and G. A. Lanzarone. Metalevel negation in non-monotonic reasoning. *Intl. J. of Methods of Logic in Computer Science*, 1:111–140, 1994.

38. S. Costantini and G. A. Lanzarone. A metalogical programming approach: Language, semantics and applications. *Int. J. of Experimental and Theoretical Artificial Intelligence*, 6:239–287, 1994.

39. S. Costantini, G. A. Lanzarone, and L. Sbarbaro. A formal definition and a sound implementation of analogical reasoning in logic programming. *Annals of Mathematics and Artificial Intelligence*, 14:17–36, 1995.

40. F. Cuppens and R. Demolombe. Cooperative answering: a methodology to provide intelligent access to databases. Proc. 2nd Int. Conf. on Expert Database Systems, pages 621–643, Virginia, 1988.

41. D. De Schreye and B. Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation. In A. Pettorossi, editor, *Meta-Programming in Logic*, LNCS 649, pages 192–204, Berlin, 1992. Springer-Verlag.

42. P. Dell'Acqua. *SLD-Resolution with Reflection.* PhL Thesis, Uppsala University, Uppsala, 1995.

43. P. Dell'Acqua, F. Sadri, and F. Toni. Combining introspection and communication with rationality and reactivity in agents. To appear in: *Logic in Artificial Intelligence* (Jelia'98), LNAI, Springer-Verlag, 1998.

44. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam, 1990.

45. Pierre Cointe E. des Mines de Nantes et al., editor. *OOPSLA 1993 Workshop on Reflection and Meta-level Architectures*, Washington D.C., 1993.

46. K. Eshghi. *Meta-Language in Logic Programming.* PhD thesis, Dept. of Computing, Imperial College, London, 1986.

47. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new declarative semantics for logic languages. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Intl. Conf. Symp. on Logic Programming*, pages 993–1005, Cambridge, Mass., 1988. MIT Press.

48. S. Feferman. Transfinite recursive progressions of axiomatic theories. *J. Symbolic Logic*, 27:259–316, 1962.

49. S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III*, pages 21–36, Berlin, 1996. Springer-Verlag.

50. L. Fribourg and F. Turini, editors. *Logic Program Synthesis and Transformation – Meta-Programming in Logic*, LNCS 883. Springer-Verlag, 1994.

51. N. E. Fuchs and M. P. J. Fromherz. Schema-based transformations of logic programs. Proc. Workshop Logic Program Synthesis and Transformation, 1992.

52. T. H. Fung and R. Kowalski. The IFF proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–165, 1997.

53. H. Gallaire and C. Lasserre. Metalevel control for logic programs. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 173–85. Academic Press, London, 1982.

54. M. R. Genesereth and S. P. Ketchpel. Software agents. *Comm. ACM*, 37(7):48–53, 1994.

55. M. Haraguchi and S. Arikawa. Reasoning by analogy as a partial identity between models. In K. P. Jantke, editor, *Analogical and Inductive Inference*, LNCS 265, pages 61–87, Berlin, 1987. Springer-Verlag.

56. J. Harrison. Metatheory and reflection in theorem proving: a survey and critique. Technical report, University of Cambridge Computer Laboratory, 1995.

57. C. Higgins. On the declarative and procedural semantics of definite metalogic programs. *J. Logic and Computation*, 6(3):363–407, 1996.

58. P. M. Hill and J. Gallagher. Meta-programming in logic programming. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5.* Oxford University Press, 1995.

59. P. M. Hill and J. W. Lloyd. Analysis of metaprograms. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–51, Cambridge, Mass., 1988. MIT Press.

60. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

61. URL: http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm.

62. G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω.* PhD thesis, Université Paris VII, Paris, 1976.

63. J. Jaffar, J.-L. Lassez, and M. J. Maher. A theory of complete logic programs with equality. *J. Logic Programming*, 3:211–223, 1984.

64. J. Jaffar, J.-L. Lassez, and M. J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming–Functions, Relations, and Equations*, pages 441–467. Prentice-Hall, Englewood Cliffs, N.J., 1986.

65. Y. J. Jiang. Ambivalent logic as the semantic basis of metalogic programming: I. In P. Van Hentenryck, editor, *Proc. 11th Intl. Conf. on Logic Programming*, pages 387–401, Cambridge, Mass., 1994. MIT Press.

66. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. Logic and Computation*, 2(6):719–770, 1993.

67. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, UK, 1998.

68. M. Kerber. Some aspects of analogy in mathematical reasoning. In K. P. Jantke, editor, *Analogy and Inductive Inference, Proc. Int. Workshop AI'89*, LNAI 397, pages 231–242, Berlin, 1989. Springer-Verlag.

69. J. S. Kim and R. A. Kowalski. A metalogic programming approach to multi-agent knowledge and belief. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

70. R. A. Kowalski. Predicate logic as a programming language. In J. L. Rosenfeld, editor, *Information Processing, 1974*, pages 569–574, Amsterdam, 1974. North-Holland.

71. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases, Intl. Workshop LID'96*, LNCS 1154, pages 137–149, Berlin, 1996. Springer-Verlag.

72. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

73. K. Kunen. Negation in logic programming. *J. Logic Programming*, 4:289–308, 1987.

74. C. S. Kwok and M. Sergot. Implicit definition of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Intl. Conf. Symp. on Logic Programming*, pages 374–385, Cambridge, Mass., 1988. MIT Press.

75. Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 234–242, San Francisco, 1997. Morgan Kaufmann.

76. D. Lankford. Canonical inference. Technical Report ATP-32, Dept. of Mathematics and Computer Science, Austin (Texas), 1975.

77. G. A. Lanzarone. Metalogic programming. In M. I. Sessa, editor, *1985–1995 Ten Years of Logic Programming in Italy*, pages 29–70. Palladio, 1995.

78. G. Levi and D. Ramundo. A formalization of metaprogramming for real. In David S. Warren, editor, *Logic Programming – Proc. 10th Intl. Conf. on Logic Programming*, pages 354–373, Cambridge, 1993. MIT Press.

79. J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, Berlin, 1987.

80. T. M. Lownie. Extending reflective architectures. In *Proc. 17th Intl. Joint Conf. on Artificial Intelligence*, pages 446–451, Los Altos, Calif., 1989. Morgan Kaufmann.

81. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4:258–282, 1982.

82. B. Martens and D. De Schreye. Why untyped nonground metaprogramming is not (much of) a problem. *J. Logic Programming*, 22, 1995.

83. A. Mendhekar and D. Friedman. An exploration of relationships between reflective theories. In G. Kiczales, editor, *Proc. Reflection '96.* Xerox PARC, 1996.

84. R. Moore. Reasoning about knowledge and action. In *Proc. Fifth Intl. Joint Conf. on Artificial Intelligence*, pages 223–227, Los Altos, Calif., 1977. Morgan Kaufmann.

85. P. M. Nardi and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, Amsterdam, 1988.

86. H. S. Nwana. Software agents: an overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

87. H. S. Nwana and D. T. Ndumu. An introduction to agent technology. In H. S. Nwana and N. Azarmi, editors, *Software agents and soft computing*, LNAI 1198, pages 3–26, Berlin, 1996. Springer-Verlag.

88. D. Pastre. Muscadet: An automatic theorem proving system using knowledge and metaknowledge in mathematics. *Artificial Intelligence*, 38:257–318, 1989.

89. M. Pattie. Artificial life meets entertainment: Life like autonomous agents. *Comm. ACM*, 38:108–114, 1995.

90. D. Perlis. Languages with self-reference I: Foundations (or: We can have everything in first-order logic!). *Artificial Intelligence*, 25:301–322, 1985.

91. D. Perlis and V. S. Subrahmanian. Meta-languages, reflection principles, and self-reference. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. II: Deduction Methodologies.* Oxford University Press, 1994.

92. A. Pettorossi, editor. *Meta-Programming in Logic*, Berlin, 1992. Springer-Verlag.

93. F. Pfenning. The practice of logical frameworks. In H. Kirchner, editor, *Trees in Algebra and Programming - CAAP '96*, volume 1059 of *LNCS 1059*, pages 119–134, Linkoping, Sweden, 1996. Springer-Verlag.

94. G. Polya. *How to solve it.* Princeton University Press, 1949.

95. W. V. Quine. *Mathematical Logic.* Harvard University Press, Cambridge, Mass., 1947.

96. A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-92)*, pages 473–484. Morgan Kaufmann, 1991.

97. A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In V. Lesser, editor, *Proc. First Intl. Conf. on Multi-Agent Systems*, pages 312–319. MIT Press, 1995.

98. J. A. Robinson. Fast unification (abstract). In *Tagung über Automatisches Beweisen.* Mathematisches Forschungsinstitut Oberwolfach, 1976.

99. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall, 1995.

100. T. Sato. Meta-programming through a truth predicate. In K. Apt, editor, *Proc. Joint Intl. Conf. Symp. on Logic Programming 1992*, pages 526–540, Cambridge, Mass., 1992. MIT Press.

101. P. Schroeder-Heister. Hypothetical reasoning and definitional reflection in logic programming. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, LNAI 475, pages 327–339, Berlin, 1991. Springer-Verlag.

102. J. R. Searle. *Speech Acts.* Cambridge University Press, 1969.

103. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

104. M. P. Singh. Towards a formal theory of communication for multiagent systems. In *Proc. 12th Intl. Joint Conf. on Artificial Intelligence*, pages 69–74, Sydney, Australia, 1991. Morgan Kaufmann.

105. B. Smith and A. Yonezawa, editors. *Proc. of the IMSA'92 Int. Workshop on Reflection and Meta-level Architectures.* Research Institute of Software Engineering, 1992. Tokyo.

106. B. C. Smith. Reflection and semantics in a procedural language. Technical report, MIT MIT/LCS/TR-272, Cambridge (MA), 1982.

107. B. C. Smith. Varieties of self-reference. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Theoretical Aspects of Reasoning about Knowledge*, pages 19–43. Morgan Kaufmann, 1986.

108. E. P. Stabler. Representing knowledge with theories about theories. *J. Logic Programming*, 9:105–138, 1990.

109. L. Sterling and E. Y. Shapiro, editors. *The Art of Prolog*. MIT Press, Cambridge, Mass., 1986.

110. V. S. Subrahmanian. Foundations of metalogic programming. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 1–14, Cambridge, Mass., 1988. MIT Press.

111. P. Suppes. *Introduction to Logic*. Van Nostrand Reinhold Company, New York, 1957.

112. A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its applications to metaprogramming. In H. J. Kugler, editor, *Information Processing 86*, pages 415–420. North-Holland, 1986.

113. A. Tarski. The concept of truth in formalized languages. In *Logic, Semantics, Metamathematics*, pages 152–278. Clarendon Press, Oxford, 1956.

114. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

115. F. van Harmelen. Meaningful names: formal properties of meta-level naming relations. deliverable of ESPRIT Basic Research Project 3178 (REFLECT), 1990.

116. F. van Harmelen. Definable naming relations in meta-level systems. In A. Pettorossi, editor, *Meta-Programming in Logic*, LNCS 649, pages 89–104, Berlin, 1992. Springer-Verlag.

117. F. van Harmelen, B. Wielinga, B. Bredeweg, G. Schreiber, W. Karbach, M. Reinders, A. Voß, H. Akkermans, B. Bartsch-Spörl, and E. Vinkhuyzen. Knowledge-level reflection. In *Enhancing the Knowledge Engineering Process – Contributions from ESPRIT*, pages 175–204. Elsevier Science, Amsterdam, The Netherlands, 1992.

118. B. van Linder, W. van der Hoek, and J.-J. Ch. Meyer. Communicating rational agents. In B. Nebel and L. Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence*, LNAI 861, pages 202–213, Berlin, 1994. Springer-Verlag.

119. E. Verharen and F. Dignum. Cooperative information agents and communication. In P. Kandzia and M. Klusch, editors, *Cooperative Information Agents*, LNAI 1202, pages 195–208, Berlin, 1997. Springer-Verlag.

120. G. Wagner. Multi-level security in multiagent systems. In P. Kandzia and M. Klusch, editors, *Cooperative Information Agents*, LNAI 1202, pages 272–285, Berlin, 1997. Springer-Verlag.

121. R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, pages 133–70, 1980.

122. P. Winston. Learning and reasoning by analogy. *Communication of the Association for Computing Machinery*, 23:689–703, 1980.

123. M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: a survey. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*, pages 1–22, Berlin, 1995. Springer-Verlag.

124. T. Yokomori. Logic program forms. *New Generation Computing*, 4:305–309, 1986.

Index