Preface

Background

The term meta-programming refers to the ability of writing programs that have other programs as data. It is usual to refer to the programs that play the role of data as *object programs*, and to the manipulating programs as *meta-programs*. To be more precise, meta-programs work on a *representation* of object programs.

Meta-programming has played a fundamental role both in the foundations of computer science and in its practical developments. Its roots go back to mathematical logic, more specifically to Kleene's normal form theorem that states that for some primitive recursive functions T, U every partial recursive function equals $U(\mu y.T(e, x, y))$ (usually denoted by ϕ_e) for some natural number e. We can view ϕ_e as the function computed by the program e, and U and T as meta-programs that work on the object program e.

The choice of logic programming as a basis for meta-programming offers a number of practical and theoretical advantages. One of them is the possibility of tackling critical foundational problems of meta-programming within a framework with a strong theoretical basis. Another is the surprising ease of programming.

However, to formally deal with meta-programs, the usual framework of logic programming, or more generally, of first order logic has to be modified and appropriately extended. The reason is that various phenonemena relevant to meta-programming, like the representation of object programs and their syntax, the interplay between the object level and meta-level, the use of modules, the representation of proof strategies etc. require logics that are richer and more expressive.

We denote these extensions of logic programming and first order logic collectively by the term of *meta-logics*. Their definitions, formal properties and use form the main theme of this book. The other is meta-programming in logic programming.

Let us discuss now some of the issues mentioned above in more detail. This will provide us with a better insight into the problems that need to be solved on the meta-logical level.

From the viewpoint adopted above, it is certainly true that compilers can be considered as meta-programs in their full right. But, is it reasonable to consider a text editor as a meta-program, when it is applied to a file containing the representation of a program? In this case the most sensible answer seems to be "no", and, in order to cope with this contradiction, we propose to refine our informal definition into: *meta-programming refers to the ability of writing programs that have other programs as data and exploit their semantics.*

Among the examples of useful meta-programs we can list compilers, interpreters, program analyzers, and partial evaluators.

In the application field, things get even more interesting when the meta-program and the object program are written in the same language. The deep and far reaching phenomenon that arises in this case is that the very same piece of syntax can play, in principle, either the passive role of datum or the active role of program construct in different contexts and in different stages of the program execution.

In Prolog, the minimal example of this phenomenon is represented by the clause

```
eval(x) :- x
```

that allows a programmer to "lift" any ordinary term (a Prolog datum) to the status of being an atom (a Prolog programming construct).

Such a double role for the same piece of syntax has been typical for many knowledge representation systems that have been realized and experimented with.

The idea is simple. Write a program that takes as input the representation of a program written in a superset of the language, and execute it according to the following strategy: if a construct is a new one then call a special procedure to handle it, otherwise let the construct be executed as it is. Many books on the use of Lisp or Prolog for artificial intelligence applications contain extensive examples of this approach. What these books lack, however, is a justification of this approach from the logical point of view.

The most critical problem of meta-programming is certainly the *representation problem*, i.e. how object programs are represented within meta-program. The first part of the volume deals with this essential problem and discusses various meta-logical solutions to it. The second part of the volume is concerned with language extensions that make meta-programming easier and more elegant. Finally, the third part of the volume deals with the use of meta-logics for advanced knowledge representation purposes. Let us discuss now the individual contributions to this volume.

Part I: Foundations

A classical problem in the foundations of meta-logic programming is the justification of the formally incorrect (untyped) Vanilla meta-interpreter, which uses a non-ground representation of object variables. In particular, the unwell-typedness of Vanilla leads to the presence of unrelated atoms in the least Herbrand model of the Vanilla meta-interpreter.

The paper of **Kalsbeek** overviews and compares various approaches towards the problem of the occurrence of unrelated atoms in the semantics of the Vanilla meta-interpreter: the use of the (correct) typed version, restriction to language independent object programs, and the use of S-semantics. In particular, Kalsbeek argues that Hill and Lloyd's

seminal procedural correctness result for the typed version Vanilla is also a proof for the procedural correctness for the untyped version of the Vanilla meta-interpreter. It is also shown that the various correctness proofs are insensitive to the precise representation of the object level language. In addition, she discusses the use of ambivalent syntax as the underlying syntax for the Vanilla meta-interpreter, in particular for amalgamated extensions and enhanced versions. She presents a separate proof for the declarative correctness of the Vanilla meta-interpreter with ambivalent syntax as the underlying language. This result is then used to prove the correctness of a simple amalgamation of the object program with the associated Vanilla meta-program.

While logic programming is formally based on first order predicate logic, many of its applications use non-standard syntaxes, which are characterised by syntactical ambivalence between formulas, terms, predicates, and functions. Examples are the meta-variable facility of Prolog, the overloading of predicate and function symbols allowed in Prolog, the identity naming of object level constructs used in Vanilla meta-programming, and the use of generic predicates in databases.

The paper of **Kalsbeek** and **Jiang** discusses Ambivalent Logic, which provides a general framework for first order predicate logic with various levels of syntactic ambivalence. A conservativity result shows that Ambivalent Logic is a conservative extension of first order predicate logic. They prove a series of results which justify the use of ambivalent syntax in logic programming. In particular, they prove termination and correctness of an appropriate version of the Martelli-Montanari unification algorithm, and show that resolution is a sound and complete inference method for Ambivalent Logic.

The two best known semantics for definite logic programs are least Herbrand semantics and S-semantics. It is however not a priori clear that these semantics lead to meaningful results for meta-programs in the Prolog-style non-typed tradition, using a non-ground representation for object level variables like the well-known vanilla meta-interpreter. Since this style of meta-programming seems to be of considerable practical importance, this situation must be judged unsatisfactory.

In their contribution, **Martens** and **De Schreye** study the relation between the semantics of definite object programs and the corresponding untyped vanilla meta-programs, both in the context of least Herbrand and S-semantics. They also investigate various enhanced meta-programs, some of which feature limited forms of amalgamation. The latter extension is enabled through the overloading of function and predicate symbols, a technique that essentially coincides with allowing a certain degree of syntactical ambivalence in the language. For these programs and semantics, they establish under which conditions there is a strong correspondence between object and meta-level semantics, thus shedding light on the question to what extent meta-programming of this kind can be judged meaningful.

Also the paper by **Brogi** and **Turini** addresses the representation problem. In their contribution they propose a semantic justification for a simple representation technique in the field of a generalised notion of meta-programming in logic. The representation technique is again based on the notion of ambivalent syntax, and the generalisation consists in specifying the meta-programs with respect to object programs defined via program expressions. The expressions are defined by means of a rich suite of operations on logic programs. The technique allows one to build straightforward and concise meta-programs via the representation of object level variables by meta-level variables.

One of the interesting features of Prolog is that it allows us to extend its syntax in a simple way, by means of meta-variables. This property is used to define negation in Prolog, using meta-variables, clause ordering and the cut operator. In logic programming negation is defined in quite a different way, by means of so-called SLDNF-resolution.

The paper by **Apt** and **Teusink** compares these two uses of negation – in Prolog and in logic programming. This requires a careful reexamination of the assumptions about the underlying syntax and a precise definition of the computational processes involved. After taking care of these matters, among others by adopting an ambivalent syntax, they prove an equivalence in appropriate sense between these two uses of negation. This result allows them to argue about correctness of Prolog programs that use negation.

Part II: Language Support for Meta-Logics

High-level languages such as Lisp and Prolog are often chosen because the syntactic similarity between programs and data makes it very convenient to write meta-programs in those languages. If we desire a truly declarative programming language however, it can be seen that we have been tempted down a blind alley in the approach these languages take to meta-programming, because they do not provide the means for a declarative treatment of object variables in the meta-program. Without a ground representation, any but the simplest of meta-programs can have no declarative semantics. Unfortunately, using a ground representation apparently incurs a significant overhead in program complexity and computation time.

Gödel is a new programming language aimed at narrowing the gap between theory and practice in logic programming, and with particular emphasis on declarative meta-programming. To achieve its aim, Gödel must make the ground representation attractive to programmers in both ease of use and execution time. The paper by **Bowers** and **Gurr** shows how this might be done, through the careful design of library modules, and the use of partial evaluation and low-level implementation techniques. Their experiments on simple Gödel meta-interpreters using the ground representation demonstrate some

dramatic performance improvements from these methods.

The paper by **Brogi** and **Contiero** investigates the adequacy of Gödel as a meta-language for implementing various forms of logic program composition. Two alternative implementations of a set of program composition operations are presented, based on the non-ground and the ground representation of object programs respectively. The merits of Gödel as a meta-language are discussed by comparing the two implementations and by analysing the results of some experiments with the Gödel partial evaluator. Finally, some directions in which Gödel might be extended or improved are identified.

In the paper by **Barklund**, **Boberg**, **Dell'Acqua**, and **Veanes** *theory systems* are proposed as a device for writing software engineering applications and applications that involve reasoning and meta-reasoning. A theory is a set of sentences that is closed under inference and a theory system is then a collection of theories that are related through reflection principles.

The meta-logic programming language *Alloy* for defining theory systems is introduced with formal syntax, inference rules and a concept of models for Alloy programs. Several examples of Alloy programs that define theory systems are given.

Part III: Meta-Logics for Knowledge Management

Traditional logic is concerned with static theories, which do not change over the course of time. Deductive databases and knowledge bases extend this static form of logic to include the dynamics of database updates and knowledge assimilation. Such dynamic theories, however, are still essentially passive, in that, although they change their own internal state, they do not change the state of the environment.

Kowalski's paper proposes the use of meta-logic programming, within a concurrent logic programming framework, to extend such theories to active theories, which behave as intelligent agents. He presents a meta-logic program which defines the observation-thought-action cycle of such an agent, with the intention of giving the definition both a procedural (process) and declarative (logical) interpretation. Moreover, he argues that, by adjusting the amount of resources available for thought versus observation and action, it is possible to simulate both reactive agents (when the amount of resource is small) and rational agents (when the amount is sufficiently large).

In knowledge representation several formalisms for reasoning about knowledge in a multi agent scenario have been proposed. More specifically, we can identify a family of languages based on the use of a modal operator and another one based on the use of first-order logic enriched with meta-level capabilities.

The paper by **Carlucci Aiello**, **Cialdea**, **Nardi** and **Schaerf** considers these two

approaches by addressing the issues of consistency that arise from selfreferentiality, their expressiveness and the methods for translating classical modal systems into meta-level first-order formalisms.

Preferences and strategies are fundamental to model-based diagnosis, for specifying preferred and fall-back approaches to the diagnosis task, both to capture general and domain specific criteria, but also to tackle the complexity issue by employing heuristics.

The paper by **Damásio**, **Nejdl**, **Pereira**, and **Schroeder** presents a formal framework based on extended logic programming and meta-programs for the representation of preferences and strategies required by model-based diagnosis. This framework is clearer and more expressive than other approaches that have addressed these problems. The authors show how the concepts of preferences and strategies are directly programmed and captured by logic meta-programming and meta-reasoning methods, and their implementation techniques. The paper is intended as proof-of-principle that all concepts needed by a model-based diagnosis system can represented declaratively and captured by a logic meta-program. Specialized more efficient algorithms can be substituted for the simpler proof-of-principle ones they include, and are the subject of ongoing work.

Meta-programming can also be used as a theoretical basis for defining more expressive data models. The paper by **Sripada** and **Möller** describes a rich temporal data model for advanced database applications. They illustrate the power of meta-programming for temporal knowledge representation and reasoning. They then describe how the relational model can be extended to provide support at the database level for the concepts derived from metalevel knowledge representation.

Acknowledgments

This volume presents an outcome of research carried out within the Esprit funded Basic Research Project "Compulog II". (For those interested in numbers - No. 6810.) The project has started August 1, 1992 and will finish July 31, 1995. The first editor is the project coordinator and the second one coordinator of the area "Meta- and non-monotonic reasoning" within the project.

The aim of "Compulog II" has been to study various extensions of logic programming which make it more amenable for knowledge representation and programming. One of the important elements has been the investigation of the meta-programming within the logic programming paradigm. This book is devoted to this aspect of Compulog II research. Many chapters were specially written for this book. They were all internally refereed. We would like to take this opportunity to thank all the authors for their contributions, refereeing work and assistance in preparing this preface. Bob Prior from the MIT Press provided us with the necessary help on the side of the publisher and, last but not least, Kees Doets and Bonnie Friedman helped us to win in our struggle with the MIT stylefiles.

We hope this volume will not satisfy but rather stimulate readers' interest in this exciting research area.

K.R.A.
F.T.