Frank van Harmelen
Department of Artificial Intelligence
University of Edinburgh

Meta-level Inference Systems

Pitman, London

Morgan Kaufmann Publishers, Inc., San Mateo, California

Contents

Abstract

In this book we will be concerned with a particular type of architecture for reasoning systems, known as meta-level architectures.

The book is divided in two parts. The first part discusses general properties of meta-level systems, whereas the second part concentrates on a particular efficiency problem associated with meta-level systems.

After presenting the arguments for meta-level systems (chapter 1), we discuss a number of systems in the literature that provide an explicit meta-level architecture (chapter 2), and these systems are compared on the basis of a number of distinguishing characteristics. This leads to a classification of meta-level architectures (chapter 3). Within this classification we compare the different types of architectures, and argue that one of these types, called bilingual meta-level inference systems, has a number of advantages over the other types. We study the general structure of bilingual meta-level inference architectures (chapter 4), and we discuss the details of a system that we implemented which has this architecture (chapter 5). One of the problems that this type of system suffers from is the overhead that is incurred by the meta-level effort. We give a theoretical model of this problem, and we perform measurements which show that this problem is indeed a significant one (chapter 6). Chapter 7 discusses partial evaluation, the main technique available in the literature to reduce the meta-level overhead. This technique, although useful, suffers from a number of serious problems. We propose another technique, that we baptise partial reflection (chapter 8), which can be used to reduce the problem of meta-level overhead without suffering from these problems. Chapter 9 concludes and discusses future research directions.

Preface

This book is a slightly revised version of the dissertation I submitted for a Ph.D. degree at the Department of Artificial Intelligence of the University of Edinburgh in 1989.

Parts of the content of this book have been published elsewhere, and, to avoid the scientific sin of overpublication, the question must be answered what this book contributes over and above the already published material.

To be precise the following material has been published previously elsewhere:

- an early version of chapter 3 has been published as a journal article under the title "An overview of meta-level architectures for control in expert systems" in the *Journal of Information Processing and Cybernetics*, Vol. **25**, No. 1, (January 1989), pp. 21–36.

- A later version of chapter 3 has appeared as "A classification of meta-level architectures", in the Proceedings of the META88 Workshop on Meta-programming in Logic-programming, University of Bristol, June 1988, pp. 103–122, published by MIT Press in 1989 under the title *"Meta-programming in Logic-programming"* H. Abramson and M.H. Rogers (eds.), ISBN 0-262-51047-2.

- Material from chapter 5 has been published as chapter 3 of the book *"Logic-based knowledge representation"*, edited by P. Jackson, H. Reichgelt and F. van Harmelen, MIT Press, 1988, and also (in a shorter form) as a journal article entitled "Socrates: A flexible toolkit for building logic based expert systems" in *The Knowledge-Based Systems Journal*, Vol. **1**, No. 3, July 1988, pp. 132–142.

- The chapter on partial evaluation, chapter 7, was an invited contribution to the Proceedings of the 5th IMYCS, published as *"Machines, Languages and Complexity"*, J. Dassow and J. Kelemen (eds), Springer Verlag Lecture Notes in Computer Science No. 381, pp. 170–187, 1989.

The reasons why I believe that the publication of this book is still warranted are threefold. First of all, some material in this book has not been published in any accessible form before, notably the material in chapters 4, 6, 8 and parts of chapter 5. Secondly, some material would simply not be publishable in separate papers, but only in the context of surrounding material. In particular, the attempt at defining some terminology in chapter 1, the literature review in chapter 2 and pointers to future research in chapter 9. Thirdly, I hope that putting all the material together in one cover will make apparent some interesting connections that would otherwise not be visible.

Finally, a remark on the intended readership of this book. As with any research material, this is not a book for beginners. The reader is assumed to be familiar with the fundamental concepts and issues in fields like knowledge representation and (to a minor extent), logic and logic programming. As a result, the book is primarily directed to fellow researchers in these fields, although some parts (notably chapters 2, 3 and the first section of chapter 7) may be of interest to advanced students.

Acknowledgements

This book is based on the work I did for my Ph.D. thesis in the Department of Artificial Intelligence of the University of Edinburgh.

I am very grateful to the following people, without whose support and contributions I could not have written done this work:

- Peter Jackson for inviting me to work in Edinburgh, and thereby getting it all started,
- Han Reichgelt for much encouragement and guidance during the first $2\frac{1}{2}$ years, which are always the most difficult,
- Fausto Giunchiglia for being a very precise and critical reader of my thesis, providing me with many valuable suggestions on content and presentation,
- Alan Bundy for being a such a good supervisor during my final year,
- Lynda Hardman for all the support, encouragement, and many long discussions,
- and all the members of the AI Department in Edinburgh for providing a wonderfully good place to work.

Chapter 1

Introduction

The goal of this chapter is to introduce the reader to the problems that we will address in this book, to introduce some essential assumptions and terminology, to make some brief methodological remarks, and to outline the structure of the book. First of all we will list the questions that we try to answer in this book. Although each of these questions will be discussed in more detail in further chapters of the book, their brief mention here will give the reader a general idea of what our goals and intentions are.

1.1 Goals of this book

Our central interest in this book will be the *architecture of reasoning systems*. The most successful type of reasoning systems so far has been the so called *knowledge-based systems* (or *expert systems*), reasoning systems in very constrained and well structured domains. As a result, we will often look at knowledge-based systems architectures, although these are not our single concern. One aspect of the architecture of reasoning systems that will interest us in particular in this book is the *control* of the reasoning process, and the way in which the architecture of a system can be adapted to help control the reasoning process. In particular, the so called *meta-level architectures* have become very popular for this purpose in recent years. In this book we are interested in acquiring a better understanding of the structure of meta-level architectures and investigating some problems associated with them. More specifically, the questions we will try to answer are:

- Which kinds of meta-level architectures have been proposed in the literature for controlling the inference process?
- How can these different architectures be compared, and which ones are best suited for controlling the reasoning process? This will lead us to identify one particular type of architecture, the so called *meta-level inference systems*.
- What are the essential components of such meta-level inference systems?
- One important problem associated with meta-level inference systems is often mentioned in the literature, namely the problem of *meta-level overhead*. What are the contributing factors to this problem, and what is the significance of the problem?

- Which possible solutions can we find to solve the problem identified above?

The main results achieved in this book are the following:

- We give a classification of existing meta-level architectures. This classification is new compared to existing classifications in the literature because it concentrates on the distribution of activity and the communication between the two essential layers in any meta-level architecture.
- We have constructed a working and realistic implementation of a logic based meta-level inference system which has been used in practical applications.
- We extend an existing analysis of the components of a meta-level architecture to apply in more detail to logic based meta-level inference systems.
- We provide both a theoretical model and experimental measurements for estimating the size of the problem of the overhead of meta-level interpretation.
- We give a number of solutions (and proposals for solutions) to this problem of meta-level overhead.

1.2 Assumptions and terminology

As with any research we will build on the work of other researchers, and use their results as our starting points. We will briefly state some conclusions and consensus reached on certain issues, without arguing these conclusions in detail. We will adopt these results as assumptions for our own work, and refer to places in the literature where the case for these assumptions is made. The results concerning the architecture of reasoning systems that we will adopt as assumptions in the rest of this work are the following:

- a distinction should be made between domain knowledge and control knowledge (the terms "domain knowledge" and "control knowledge" will be defined below)
- control knowledge should be explicitly represented (we will also discuss the meaning of the term "explicit representation")
- meta-level architectures are a good vehicle for such an explicit representation of control knowledge (we will also discuss the meaning of the term "meta-level")
- logical languages should be used as the knowledge representation languages of a reasoning system.

These assumptions, and their essential terminology, will be the main subject of the remainder of this chapter. After this discussion of assumptions and terminology we will briefly discuss a methodology for Artificial Intelligence which will make clear the position of this work in the wider field of AI. We will conclude this chapter by outlining the structure of the rest of the book.

1.2.1 Domain knowledge and control knowledge

A practice that has been well established in the last 5 to 10 years of building reasoning systems is the distinction between two types of knowledge, often called *domain knowledge* and *control knowledge*. We will not argue the virtue of such a distinction here, but instead refer the reader to the arguments that can be found in a well established body of literature, e.g. [Davis, 1980, Clancey, 1983a, Erman et al., 1984, Clancey, 1985]. Unfortunately, no formal and precise definition of this distinction is available, but we will try to clarify this distinction informally.

The terms "domain knowledge" and "control knowledge" are often used to distinguish *what* a system knows from *how* the system uses what it knows. Domain knowledge is taken to consist of descriptions of objects in the domain of expertise of the reasoning system, such as their properties and relations, typical features, known facts etc. Control knowledge is then taken to refer to strategies for the reasoning process, such as orderings of tasks in the domain, plans for achieving certain goals, encodings of general problem solving skills etc.

One problem with this terminology is that it suggests that control knowledge is domain independent (since it is different from domain knowledge). However, this is not the case, since control knowledge can be either domain dependent (i.e. specific to one domain), or domain independent (not specific to one domain)[1].

Other terminology has been used to distinguish domain knowledge from control knowledge, notably the terms *declarative* and *procedural*, where domain knowledge is declarative, and control knowledge is procedural in nature. However, this is also rather unfortunate terminology, since the terms "declarative" and "procedural" refer to the format in which knowledge has been written down, and not to its contents. Although it is true that certain formats are more or less suited for different kinds of knowledge, both domain knowledge and control knowledge can be formulated in either a declarative or a procedural form: domain knowledge is often written declaratively (e.g. logic), but can also be written procedurally (e.g. production rules), while control knowledge, although often written procedurally, can also be formulated declaratively (see the descriptions of TEIRESIAS and GOLUX in the next chapter for examples of procedural and declarative formats for control knowledge).

Confusion between domain knowledge and control knowledge often arises because part of the domain knowledge is about the order in which certain tasks should be performed to achieve a particular goal in the domain (e.g. the order in which tests should be performed on a patient to establish a diagnosis). Such knowledge refers to objects and relations in the domain of expertise, and should therefore be regarded as domain knowledge. We propose that the term control knowledge is only used for knowledge which tells the system how to use the domain knowledge. As a result, control knowledge never refers to objects, relations, facts, rules, procedures etc. of the domain of expertise (as domain knowledge does), but it only refers to elements of the domain knowledge (and in particular it is concerned with how to use the domain knowledge). This distinction can be quite subtle, as the following two rules illustrate. A rule like "use inexpensive blood-tests before expensive blood-tests" is part of the domain knowledge, since it refers to elements in the domain of expertise of the

[1] We use "domain independence" here in the sense of [Clancey, 1983a]: domain independence does not mean that the knowledge applies to every domain, but just that it is not specific to any one domain.

reasoning system (e.g. "blood-tests"), but a rule like "use rules that mention inexpensive blood-tests before rules that mention expensive blood-tests" is part of the control knowledge, since it refers to items of the domain knowledge, and how to use them. A major advantage of these definitions is that they allow us to make a *syntactic* distinction between the two types of knowledge, in other words: we can classify an expression as domain or control knowledge purely by looking at the things that occur in the expression.

Using the terms domain knowledge and control knowledge in this way, we have to keep in mind that control knowledge can be either domain dependent or domain independent, depending on whether it refers to the contents of particular elements of domain knowledge, as in "use rules that mention cheap blood-tests before rules that mention expensive blood-tests", or whether it only refers to the general form of the domain knowledge, without referring to its domain-specific contents, as in "use cheap rules before expensive rules". Notice that again, this definition of domain dependent versus domain independent control knowledge gives us a *syntactic* criterion to distinguish between these two types of knowledge.

1.2.2 The explicit representation of control knowledge

To achieve full advantage of the distinction between control knowledge and domain knowledge discussed in the previous section, it is not only necessary to distinguish between the two types of knowledge, but also to represent both types *explicitly*. To explain what we mean by "explicit representation", we point out that reasoning architectures are often organised around some representation language, say L_R, plus its corresponding interpreter, and that this language L_R itself is realised in some implementation language, say L_I. The idea is then that L_R is more suitable for the purposes of the reasoning system than the (more primitive) language L_I. Expressions in L_I, (such as the definition of L_R) are fixed in the system, whereas expressions in L_R can be changed to adapt the system to a particular task. We say that the expressions in L_R are *explicitly represented* (available for inspection and modification), and that expressions in L_I are *implicitly represented* (not available for inspection and modification). We stress that by "explicit representation" we do not only mean that expressions are present in the system (i.e. available for inspection), but also that they can be changed (i.e. available for modification), and that these changes will affect the behaviour of the system. For example, in section 2.4.1 we will discuss a Prolog system where part of the control knowledge (the central interpreter loop) is available for inspection by users (or their programs), but not available for modification. In such a case we would not call this control knowledge explicitly represented.

In many systems that offer a representation language L_R, the interpretation of expressions in L_R (by which we mean the way they are used by the interpreter for L_R, in other words: the control knowledge of the system) is only implicitly represented in the system in the way the interpreter for L_R was defined in terms of L_I. In recent years, many workers in AI have argued in favour of the separate and explicit representation of control knowledge as defined above. The value of such a separate and explicit representation is one of the major assumptions underlying this work, and we will refer to some of the arguments made by other workers in AI in support of this assumption.

A system with an explicit representation of its control knowledge is easier to develop,

debug and modify, as argued in [Davis and Buchanan, 1979], [Davis, 1980], [Bundy and Welham, 1981], [Clancey, 1983a] and [Aiello and Levi, 1984]. This explicit representation enables the independent variation of control knowledge and domain knowledge. One can adopt a different strategy in dealing with a particular domain with or without changing the domain knowledge representation, and vice versa. This independence is vital: if a first-shot implementation of an expert system performs disappointingly (as is usually the case), then it is important that it is possible to (i) identify where the problem lies in terms of the architecture of the system, and (ii) make the requisite modification to the right module or module interface without having to modify other modules.

A second point is made in [Breuker and Wielinga, 1986]. They note that experts are able to use their domain knowledge for a number of different tasks, e.g. solving problems, teaching, communicating new insights, planning solutions etc. The explicit representation of control knowledge would allow the same domain knowledge to be used for a number of different purposes. Clancey [Clancey, 1985] formulates this as the possibility to write programs that interpret knowledge bases from multiple perspectives, providing the foundation for explanation, learning and teaching capabilities. A related point is mentioned in [Clancey, 1983a]. The separation of control knowledge from domain knowledge not only makes it possible to use the same domain knowledge for different purposes, but the same control knowledge can also be used in different domains[2].

An interesting side-effect of the separation of domain and control knowledge is its usefulness in knowledge elicitation. As is clear from the work of Breuker and Wielinga, a sound epistemological analysis is a prerequisite for successful knowledge elicitation. The distinction between domain knowledge of experts and the strategies experts use in employing their domain knowledge is a very useful decomposition in this context.

Fourthly, [Clancey, 1983b], [Clancey and Letsinger, 1981], [Warner Hasling et al., 1984] and [Warner Hasling, 1983] stress the importance of explicit control knowledge for the purpose of explanation. By explicitly representing the control knowledge, this part of the system can also be made subject to the explanation facilities. After all, not only should a system be able to explain what piece of knowledge was used, but also how this piece of knowledge was used and why. This enables a much deeper explanation of the behaviour of the system.

Finally, the separation of control knowledge from domain knowledge allows domain knowledge to be purely declarative in nature. While formulating domain knowledge we do not have to worry about efficiency, only about the "representational adequacy" (in the words of McCarthy). In the control knowledge on the other hand, the efficiency of the problem solving process is the most prominent aspect ("computational adequacy").

1.2.3 The use of meta-level architectures

The strong arguments given above in favour of the explicit representation of control knowledge do of course raise the question about the feasibility of this idea. Is it indeed possible to realise this idea in practical systems? To answer this question, many workers in AI have

[2]As a proviso to this point, it must be stated that there is hardly ever a full independence between domain- and control knowledge. In practice there will always be some connection between the contents of the domain knowledge and the use of (i.e. the control knowledge).

implemented architectures to support the explicit and separate representation of control knowledge. One family of architectures that is particularly natural for this purpose, and that has been particularly successful in these attempts are the so called *meta-level architectures*. A significant part of this book is devoted to describing, classifying, comparing and analysing these meta-level architectures. In this section we will more precisely describe what me mean by "meta-".

The original Greek word "$\mu\epsilon\tau\alpha$" means "beyond", as in "metaphysics" and "metamathematics", but has recently come into use as meaning "about" (possibly also as in "metamathematics"). So, meta-knowledge means "knowledge about knowledge", and "meta-level" means "a level which is about another level", and "meta-level inference" means "inference performed at one level which is about another level". In the words of Giunchiglia [personal communications], when speaking of meta-level inference, we think of a situation where there are two theories, one called the *object-level theory*, which is about a certain topic, and another theory, called the *meta-level theory*, which is about the object-level theory. The goal of object-level inference is to obtain results in the topic of interest. The goal of meta-level inference is to obtain results about the theory of the topic of interest (i.e. about the object-level theory), and then to use them to obtain better results about the topic of interest. "Better results" in this context can mean many things. It can mean "to solve the problem in less time" in which case the meta-level is used to control or guide the search at the object-level; it can mean "to get results otherwise unobtainable", in which case the meta-level inference is used to extend the solutions obtainable by the object-level; or it can mean "to describe the object-level", in which case the meta-level inference is used to get a better understanding of the object-level, which can be used for further operations (e.g. accumulating statistics about the use of the object-level knowledge, consistency and redundancy checks on the object-level knowledge, learning new object-level knowledge etc.). [Davis, 1978] and [Hayes-Roth et al., 1983] mention many of the possible uses of meta-level knowledge.

Out of all these possible uses of meta-level knowledge, in this book we will only be interested in its use as control knowledge. From the definition of control knowledge and domain knowledge in a previous section, it is clear that control knowledge can be formulated as meta-level knowledge. There, we said that

> "... control knowledge never refers to objects, relations, facts, rules, procedures etc. of the domain of expertise (as domain knowledge does), but it only refers to elements of the domain knowledge."

making control knowledge clearly meta-level knowledge, i.e. knowledge about domain knowledge (which takes the place of object-level knowledge). As we will see later in this book, control knowledge can be usefully formulated as meta-level knowledge, resulting in a meta-level theory that can be used to guide and control the inference in the object-level theory. Therefore, although meta-level knowledge should not just be equated with control knowledge (pace [Davis and Buchanan, 1979]), the discussion will focus on meta-level knowledge as control knowledge.

Thus, meta-level architectures are architectures with an explicit representation of control information. In the previous section, "explicit representation" was defined as both

inspectable and modifiable. A consequence of this definition is that whether or not a particular architecture is a meta-level architecture is dependent on the intended use of the system. In particular, the required possibility for modification depends on this intended use. For example, consider a Prolog interpreter written in C, which is itself executed by a C interpreter. For a Prolog programmer, the C program that implements the Prolog interpreter is not a meta-level interpreter for the Prolog code, since the control information encoded in the C program is neither inspectable nor modifiable. However, to a Prolog system implementer, the C program is a meta-level interpreter containing an explicit (inspectable and modifiable) representation of the procedural interpretation of an object-level program (namely a Prolog program). Thus, whenever we talk about a system being a meta-level architecture, this is always with respect to some intended use of the system, although this intended use will often be only implicitly assumed rather than actually stated.

1.2.4 Logic as a knowledge representation language

Most reasoning systems make what Smith [Smith, 1985] calls the "knowledge representation hypothesis", which says that we can construct a language for representing the knowledge embodied in the system. Many different representation languages have been used for this purpose, and in this book we will concentrate on systems that use logical languages. This choice of logic as the main formalism implies that logical languages will serve as the representational scheme, while logical deduction will be the paradigm for the inference engine. The case for logic as a representation formalism has been adequately made in other places (e.g. [Hayes, 1977], [Kowalski, 1979], and more recently [Moore, 1984]), but we will repeat the major arguments here.

First of all, unlike most other knowledge representation formalisms, logics come with a formal semantics, giving a precise description of the meaning of expressions in the formalism. Because this precise semantics allows the comparison of different logical languages, it is possible (as argued in [Reichgelt and van Harmelen, 1985, Reichgelt and van Harmelen, 1986]) to construct a set of guidelines that correlate characteristics of particular domains and tasks with appropriate logic based representational and inferential mechanisms, thereby making possible an informed choice for a representation formalism.

Secondly, again unlike many other knowledge representation formalisms, logics have well understood properties as regards their completeness, soundness, and decidability. For any reasonable logic it is possible to prove that the proof theory is sound. Furthermore, it is possible to establish via formal methods whether a particular logic is complete or not. Finally, it is known whether provability in a logic is decidable or not: for any reasonably powerful logic, provability is at best a semi-decidable property. Although these results in themselves are sometimes negative (e.g. *in*completeness, *semi*-decidability), the important point is that these properties are known at all. For many other knowledge representation formalisms no such results have been obtained.

The final, but certainly not the least, advantage in favour of logic is its expressive power. Two aspects of this must be mentioned. Firstly, the language of logic is not restricted to that of standard two-valued, truth functional, first order predicate calculus. Many other logics have been proposed, offering a wide range of expressional and inferential power. A few examples of these are intuitionistic logics, many-valued logics, modal logics,

epistemic logics and tense logics. The second important aspect of the expressive power of logic is its ability to express what might be called incomplete knowledge, or information about incompletely known situations [Moore, 1982]. As Levesque and Brachman [Levesque, 1984] put it: the expressive power of logic "determines not so much what can be said, but what can be left unsaid". As a result, one is not forced to represent details that are not (yet) known. A few examples may clarify this point. Even restricting ourselves to a comparatively restrictive logic, such as classical predicate calculus, we can express the knowledge that an object x has a given property P without knowing its identity: $\exists x P(x)$. We can also express disjunctive knowledge, such as $P \vee Q$ without stating which disjunct is true. Finally, we can draw a distinction between something not being known, i.e. P not appearing in the set of axioms, and something being known to be false, i.e. $\neg P$ appearing in the set of axioms.

1.3 Methodology

Being the relatively young science it is, AI has not yet arrived at a firm and widely accepted methodological basis. Because of this, we feel it is important to be explicit about the methodological views that underly this work, and about the role that we think this work plays in AI.

In this section we will summarise a sketch of a methodological analysis of AI as given in [Bundy, 1987], and we will then place the work done in this book in the context of that analysis, giving it a place in the broader context of AI.

Bundy distinguishes three types of AI:

- *applied* AI which is concerned with the use of existing AI techniques to build commercial, educational, military or industrial applications,
- *cognitive science* (also sometimes called *computational psychology*), which aims at the understanding and modelling of human intelligence using AI techniques, and
- *basic* AI whose goal it is to explore computational techniques which have the potential for simulating intelligent behaviour. These computational techniques can be algorithms, representation techniques or architectures.

It is in this third category that we consider this work to belong. In basic AI, researchers develop new techniques, test them and find out their interrelationships. These techniques can then be used in applied AI or in cognitive science. Part of this study is the building of computer programs embodying these techniques in order to discover them, extend them and explore their properties. Given that basic AI consists of the invention and development of techniques, Bundy identifies a number of ways by which basic AI can be progressed:

- Inventing a new technique
- Improving an existing technique
- Discovering new properties of or relations between techniques, e.g.:

- proving a technique correct or sound
- discovering the complexity of a technique
- showing that one technique is a special case of another
- showing that one technique actually consists of a number of different ones.
- showing that a number of techniques are actually the same.
- demonstrating that a technique applies to a new domain.
- exploring the behaviour of a technique on a range of standard examples.

On the basis of a number of criteria by which basic AI techniques should be judged (clarity, power, parsimony, correctness and completeness), Bundy distinguishes a four step methodology for basic AI:

1. Exploratory programming to construct a program that performs a particular task.
2. Analysis of this program to extract the essential technique(s).
3. Generalise and rationally reconstruct the technique to improve it.
4. Identify shortcomings, and try to overcome these using [1] (and so the methodology loops).

It should be clear from the above description that the work that we set out to do in this book is in the category of basic AI: the particular computational technique that we are interested in is the use of meta-level architectures for controlling inference. We are trying to advance basic AI by trying to understand this existing technique in a new way, by identifying certain problems with it, and by trying to improve it by solving these problems. Chapters 2, 3 and 4 all fall within the second methodological step (analysis of existing programs to extract the essential techniques), while chapters 5, 6, 7 and 8 fall within step [4] (identification and repair of shortcomings).

1.4 Structure of this book

As a final part of this introductory chapter, we will describe the structure of the rest of the book, thereby laying out a "path through the book" for the reader, describing the goal of each of the chapters and the way they fit together.

The book is divided in two parts. The first part discusses general properties of meta-level systems, whereas the second part concentrates on a particular efficiency problem associated with meta-level systems.

The next chapter, chapter 2, provides a guided tour among some of the important meta-level architectures in the AI literature. Each of the systems in this chapter is described "as is", without much further analysis. This analysis is the task of chapter 3. There we classify the systems described in chapter 2 on the basis of characteristic features of their architectures. This classification enables us to compare the different approaches, which leads us to argue that one particular type of architecture, the so called bilingual,

meta-level inference systems, are the most attractive. In chapter 4 this type of system is subjected to a closer analysis, and a number of essential components are identified. Chapter 5 presents a particular case study of a meta-level inference system that has been built and used and discusses some of the choices and trade-offs in such a design. Notwithstanding the advantages of meta-level systems, chapter 6 identifies one particular problem that these meta-level inference systems suffer from, namely the problem of meta-level overhead. The extra layer of computation imposed on the architecture by the meta-level interpreter can be very expensive, and can sometimes even cancel out the efficiency improvements gained by the explicit representation of the control knowledge. We investigate the size of this problem of meta-level overhead, and conclude that this overhead is indeed significant, and often larger than acknowledged in the literature. Chapters 7, and 8 discuss a number of techniques that can be used to alleviate the problem of meta-level overhead. Chapter 9 concludes and discusses remaining open problems.

Part I

Properties of architectures for meta-level inference

Chapter 2

Meta-level systems in the literature

In this chapter we discuss a number of systems from the literature that provide explicit mechanisms for controlling the inference process. This chapter is not an exhaustive survey, but is intended to contain a cross-section of the literature, presenting important representatives of a number of different approaches. The systems described below are of a very diverse nature: some are expert system shells, some are programming languages, some are expert systems applications and some are theorem provers. However, they all represent different approaches to the use of some form of meta-level reasoning for the purpose of control. The systems are all described in their own terms, using their own concepts and vocabulary. This will initially introduce a large array of apparently disconnected terminology. In the next chapter, chapter 3, we will clean up this confusion, and give a classification of meta-level systems which will enable us to compare and evaluate the different approaches.

2.1 Knowledge representation languages

In this section we describe a number of knowledge representation languages that allow the explicit representation of control knowledge as meta-level knowledge. These systems are not particular reasoning systems that solve a particular problem, but are generic systems that can be (and have been) used to build particular systems for particular applications. The first system we describe, TEIRESIAS, was one of the first systems to propose an explicit representation of control knowledge and has a rather simple meta-level architecture. The second system in this section, S1, is of a later generation, and has a much richer and diverse representation formalism. The final three systems, BB1, MLA and KRS, are from roughly the same period as S1, but rather than providing a diverse representation scheme, they aim for a single formalism that is used to represent both the domain knowledge and the control knowledge: BB1 uses a blackboard architecture for this purpose, MLA uses logic and KRS uses an object-oriented architecture.

2.1.1 TEIRESIAS

TEIRESIAS [Davis, 1980, Davis, 1982] (named after a blind seer in ancient Greece) is a system for representing knowledge in the well known production rule format. Production rules contain patterns in their left-hand side, and actions in their right-hand side. If the

patterns of a rule match against the contents of a global working memory, the rule can be fired, and the action in the right-hand side of the rule executed, possibly changing the contents of the working memory. Potentially, the left-hand side of more than one rule can be successfully matched against the working memory, creating a control problem: which of the applicable rules should be fired? This type of control problem is called *conflict resolution*, and the set of applicable rules is called the *conflict resolution set*. Many production rule systems have a hardwired, implicit conflict resolution strategy (e.g. always pick the first rule in the rule-base), or offer a limited set of built-in strategies, as in OPS5 [Waltzman, 1983], which offers a choice between a "most recent" strategy, favouring rules that match against recently added elements in the working memory, or a "most specific" strategy, preferring rules with the least number of uninstantiated variables left after the matching process.

TEIRESIAS was one of the first systems that allowed the explicit representation of the conflict resolution strategy. The choice between applicable object-level production rules is made in TEIRESIAS using meta-level production rules. Given a set of applicable object-level rules, the meta-level rules make decisions about the pruning and ordering of this set, and determine which object-level rule will be applied. This meta-level computation about the conflict resolution is done at every cycle of the object-level production rule system, that is: every time before an object-level rule is fired. Meta-rules have a limited format as illustrated in figure 2.1. Because the meta-level rules are themselves again production rules, they too potentially generate a conflict resolution problem. An interesting feature of Davis' system is that it is possible to build an arbitrarily long tower of meta-levels, where each of the levels turns to its meta-level for conflict resolution problems, until there is either no conflict resolution problem left (i.e. only one clause applies), or until there is no meta-level available any more. However, the system requires that the interpreters that are used for the various levels are the same.

2.1.2 S.1

S1 is a knowledge representation system typical of the generation of systems built after TEIRESIAS. These later systems are typically much richer in their representation formalism, offering not just one but many languages for knowledge representation within a single system. As a result they have quite different ways of dealing with their control problem. S1, developed at Teknowledge, and described in [Erman et al., 1984] is based on an epistemological analysis described in [Clancey, 1983b]. A distinction is made between structural knowledge, judgemental knowledge and control knowledge. Structural knowledge consists of a taxonomical analysis of the domain. It describes which basic objects exist in the domain, and how these objects relate to each other. This type of knowledge is often hierarchical in nature. Judgemental knowledge consists of a description of relations and properties of entities in the domain, including causal effects, known symptoms and phenomena, typical features and the like. The control knowledge is as described in section 1.2.1. The separation of the different types of knowledge in S1 is obtained by employing different representations for each type. The structural knowledge is stored in an object-oriented way. Classes are defined, and specific objects correspond to instantiations of a particular class. Information on a particular class is stored in attributes attached to the

```
PRUNING-META-RULE:
        under conditions <A> and <B>,
        rules which <do/do-not> mention <property-X>
                <at-all/
                 in-their-premise/
                 in-their-action>
        will <definitely-be-useless/
              probably-be-useless/
              ....
              probably-be-useful/
              definitely-be-useful>

ORDERING-META-RULE:
        under conditions <A> and <B>,
        rules which <do/do-not> mention <property-X>
                <at-all/
                 in-their-premise/
                 in-their-action>
        should <definitely/
                probably/
                ....
                possibly>
        be used <first/
                 last/
                 before/
                 after>
        rules which <do-not/do> mention <property-X>
                <at-all/
                 in-their-premise/
                 in-their-action>
```

Figure 2.1: format of meta-rules in TEIRESIAS

class. Judgemental knowledge is stored in standard production rule format. The control knowledge is stored in so-called control blocks. Since it is the control knowledge that has our particular interest, we will describe it in some more detail. The control blocks for expressing the control knowledge contain statements in a specialised language, containing the following primitive actions:

- Create a new object.
- Seek the value of an attribute for an object by querying the user.
- Seek the value of an attribute for an object by applying rules.

- Determine the value of an attribute for an object by using all possible means.
- Invoke another control block.
- Display text to the user.

These primitive actions can be composed using constructs such as sequencing, conditional, iterative and case statements, similar to those in conventional procedural programming languages. Within this analogy, control blocks are similar to conventional subroutines. One particular control block in an S1 knowledge base is designated the top level control block. A consultation consists of invoking this control block and performing the sequence of actions it specifies. The separation between control and judgemental knowledge is strict: no statement within a control block can directly seek the value of an attribute. Such an action can be done only by the application of rules or by querying the user, which can only be accomplished in a control block by a request for rule application or user querying. Thus, the different types of knowledge are kept separate. Notice that in this architecture, the control-blocks are "on top of" the reasoning process; they initiate subtasks such as creating a new object or determining the value of an attribute via rule-application, but they have no control over the reasoning process inside the subtask once it is initiated.

As stated above, S1 is typical of one type of knowledge representation system developed in the '80s, based on a multitude of different representation languages for different types of knowledge. Other systems of the same period have moved the opposite way, and have tried to find a single, uniform representation mechanism for both meta-level knowledge and for the different types of object-level knowledge. The final three systems discussed in this section, BB1, MLA and KRS, are all such uniform systems, although they have made different choices for their knowledge representation formalism.

2.1.3 BB1

In the blackboard architecture tradition various systems have been built to provide mechanisms for explicit reasoning about and representations of control problems. A system with a blackboard architecture is organised around a central data-structure, called the blackboard. Around this blackboard the system has a number of so called knowledge sources. Each of these knowledge sources contains a specialised portion of knowledge that is relevant to the overall problem the system is trying to solve. Each knowledge source is equipped with an input pattern, specifying the kind of problem it can solve. The central blackboard serves as a communication device between all the different knowledge sources in the following way: problem descriptions are written to the blackboard, and knowledge sources try to match their input patterns with items on the blackboard. Once a knowledge source can match its input pattern against an item on the blackboard, it becomes active. One of the active knowledge sources is then chosen for execution. This execution will lead to further information being written to the blackboard, activating or de-activating other knowledge sources, until a solution to the original problem appears on the blackboard.

Field name	Field meaning
Problem	Problem the system has decided to solve
Strategy	General sequential plan for solving the problem
Focus	Local (temporary) problem solving heuristics
Policy	Global (permanent) problem solving heuristics
To-Do-Set	Pending problem-solving activities
Chosen-Action	Problem-solving activities scheduled to execute

Figure 2.2: slots on a BB1 control blackboard

Attribute name	Attribute meaning
Description	Description of the heuristic
Goal	Predicate or function to rate potential actions desirability (0–100)
Criterion	Predicate to test for the occurrence of Goal expiration condition
Rationale	Reason for goal, e.g.: "Develop a comprehensive set of partial solutions"
Weight	Goal importance 0–1
Status	Function in control plan
Creator	Action that created this decision
Source	Information that triggered the creator
First-Cycle	Number of cycle where this heuristic was first operative
Last-Cycle	Number of cycle where this heuristic was last operative

Figure 2.3: a BB1 control heuristic

Blackboard systems can be seen as a generalisation of production rule systems, with the knowledge sources being generalisations of the production rules and the blackboard being a generalised version of the working memory. Clearly, a similar control problem arises in the blackboard system as in the production rule systems: which of the (possibly many) active knowledge sources should be chosen?

A system that tackles this problem is BB1 (**B**lack**B**oard **1**) [Hayes-Roth, 1984, Hayes-Roth, 1985]. BB1 has a dual architecture where separate blackboards and knowledge sources are available for domain reasoning and for control reasoning. BB1 expands on HEARSAY-III [Erman et al., 1981]: HEARSAY-III also provides a uniform blackboard architecture with separate, user-defined domain and control blackboards.

A simple basic control loop considers the executable knowledge sources on the domain blackboard, and schedules some of them for execution, according to the control heuristics that are recorded on the control blackboard. If the user overrides such a control decision, special control knowledge sources are triggered that engage in a dialogue with the user. This dialogue can lead to the formulation of a new strategy that will be incorporated in future actions of the system. To enable this behaviour, BB1 provides sophisticated data

structures to reason about control issues. On the control blackboard different slots for representing control decisions are available (see figure 2.2). Each of the slots on the control blackboard contains compound objects, with numerous attributes, that provide a language for control decisions. To illustrate this, figure 2.3 shows the focus slot (mentioned in figure 2.2), which contains the control heuristic currently in use. The formalism that BB1 offers to express control knowledge is much more powerful than the simple meta-level production rules of TEIRESIAS, and allows for much more elaborate control strategies, but the essential control loops of the two systems are the same.

2.1.4 MRS/MLA

A second knowledge representation system based on a uniform representation scheme is MRS (**M**eta-level **R**easoning **S**ystem) [Genesereth et al., 1980] which uses (approximately) first order predicate calculus as its representation language. An important feature of MRS is that it allows the user to annotate object-level expressions in order to specify which meta-level procedures the system should use to assert, retract or deduce that expression. For example, the statement

```
(toassert (p $x 1) fc)
```

states that to assert any statement that matches `(p $x 1)`, the system is to call the procedure `fc`. This obviously gives powerful control over the behaviour of the system. However, the weakness of the system is that the actual procedures that are used for assertion, deduction and the like (in the example above: the meaning of the procedure `fc`) are stated in implementation code (LISP), instead of in representation terms (predicate calculus). A quote from the MRS manual illustrates this:

> "..., a unique feature of MRS is that it is intended to be completely modifiable. The representation of procedures as LISP subroutines is the key to MRS's guarantee of complete modifiability..."

This quote shows a confusion about the meaning of modifiability. Of course any system is completely modifiable if the user is allowed access to the implementation of the system. However, this sort of modifiability is not what we described in the introduction, section 1.2.2. There we explained that what is required is modifiability in terms of the representation language of the system, not in terms of its implementation language. Because of all this, the best use of MRS is perhaps not as a knowledge representation system, but as an environment for the implementation of other knowledge representation systems. One such system that is implemented in MRS is MLA (**M**eta-**L**evel **A**rchitecture) [Genesereth and Smith, 1982, Genesereth and Smith, 1983]. The central idea of MLA is the use of a declarative language for describing behaviour. In this language, one can write sentences about the state of the object-level interpreter, its actions and its goals. The meta-level interpreter can reason with these sentences in deciding the ideal action for the object-level interpreter.

A key feature of the control language is that it allows partial specifications of behaviour. Presumably, when the system's behaviour is only partially specified, it relies on some hardwired control strategy for execution of the unspecified parts of the control strategy,

```
B1:     APPLICABLE(k) &
        NOT((Ex) APPLICABLE(x) & PREFERRED(x, k))
        -> RECOMMENDED(k)

B2:     OPR(k1) = ADDGOAL & OPR(k2) = ASK
        -> PREFERRED(k1, k2)

B3:     OPR(k1) = ADDGOAL & OPR(k2) = ADDGOAL &
        CF(IN(3, k1)) > CF(IN(3, k2))
        -> PREFERRED(k1, k2)

B4:     OPR(k1) = ADDGOAL &
        NUMOFSOLNS(IN(1, k1)) < NUMOFSOLNS(IN(1, k2))
        -> PREFERRED(k1, k2)

B5:     DESIRE(g) -> DEPTH(g) = 0

B6:     SUBGOAL(g1, g2, e, j) & DEPTH(g2) = n
        -> DEPTH(g1) = n+1

B7:     DEPTH(IN(1, k1)) > DEPTH(IN(1, k2))
        -> PREFERRED(k1, k2)

B8:     DEPTH(IN(1, k1)) > DEPTH(IN(1, k2))
        -> PREFERRED(k2, k1)
```

Figure 2.4: specification of search in MLA

although the authors do not explicitly say this anywhere. The language incorporates the idea of a task, which constitutes an action to be taken by the object-level interpreter. The language is that of first order logic, with the following primitive predicates:

`OPR(<k>)` designates the operation of which the task `<k>` is an instance.

`IN(<i>, <k>)` designates the `<i>`th input to the task `<k>`.

`OUT(<i>, <k>)` designates the `<i>`th output of the task `<k>`.

`BEG(<k>)` designates the start time of the task `<k>`.

`END(<k>)` designates the stop time of the task `<k>`.

`TIME(<t>)` states that `<t>` is the current time.

`EXECUTED(<k>)` states that the task `<k>` has taken place or definitely will take place.

`RECOMMENDED(<k>)` states that task `<k>` is the recommended action for a program to take.

The meta-level interpreter (i.e. the interpreter that reasons with statements in the meta-language) is programmed in LISP, and set up to reach a conclusion about `RECOMMENDED(x)`. This can be regarded as the "top level goal" of the meta-level interpreter. After a value for x has been found, the meta-interpreter has x executed by the object-level. The example in figure 2.4 shows how various types of search control can be specified. Axiom `B1` shows how ordering axioms can be used in determining which applicable task is recommended. The other axioms are examples of how this capability might be used. `B2` constrains a program to do all backward chaining before asking its user any questions, `B3` states that a program should use rules of greater certainty before rules of lesser certainty, `B4` states that, whenever a program has a choice of backchaining tasks to do, it should work on the one with fewer solutions. Axioms `B5` and `B6` define the depth of a goal in terms of its distance from an initial goal. Axioms `B7` and B8 specify depth-first and breadth-first search respectively.

2.1.5 KRS

The final knowledge representation based on a uniform representation that we will discuss is KRS (**K**nowledge **R**epresentation **S**ystem) [Steels, 1985], [Maes, 1986b,Maes, 1986a], which uses an object-oriented representation language. Knowledge is represented in objects (or concepts, as they are called in KRS). These concepts consist, as usual in object-oriented systems, of sets of slot-value pairs. The value of a slot can be interrogated or changed by sending a message to a concept. The control knowledge of an object-oriented system consists of how an object/concept is to react when it receives a message. In KRS this control knowledge is made explicit by associating a meta-concept with a concept, which specifies how the concept has to react to particular messages. Every concept in KRS has such an associated meta-concept which specifies how the object-level concept should behave. It contains the methods to make an instance of a concept, to print a concept, to inherit information in the concept, to let the concept handle messages etc. This association of meta-concepts is not restricted to one level. A meta-concept can have a meta-meta-concept and so on. Whenever something happens with a concept, the computation is handled by the meta-concept of the concept. In the example shown in figure 2.5, if the message `make-instance` is sent to concept `foo`:

```
(send foo make-instance)
```

this will result in the sending of the following message:

```
(send meta-of-foo-concept
      (how-to-respond-to-message make-instance))
```

The `meta-of-foo-concept` will compute how the concept `foo` will handle this message. When the definition of a concept does not specify what sort of meta-concept should be constructed, the concept is given a default meta-concept. Since the `meta-of-foo-concept` has no special `meta-meta-of-foo-concept`, this procedure will be executed by the default meta-concept. This default meta-concept is expressed in the implementation language of the system (LISP), and this way an infinite regress of meta-levels is stopped. Often a

concept DEFAULT-META-CONCEPT
    how-to-make-instances:
        a procedure to make an instance of the concept
    how-to-print:
        a procedure to print the concept
    how-to-respond-to-message(x):
        a procedure to let the concept respond to a message x
    how-to-inherit-information(x):
        a procedure that delegates the message x to
        the type of the concept
    ...

concept META-OF-FOO-CONCEPT
    type:
        default-meta-concept
    my-concept:
        foo
    number-of-instances-of-me:
        an integer
    how-to-make-instances:
        a procedure that makes an instance of the
        concept foo and increments the variable
        number-of-instances-of-me with one

concept FOO
    meta:
        meta-of-foo-concept

Figure 2.5: concepts and meta-concepts in KRS

specific meta-concept is defined as a specialisation of the default meta-concept, as also illustrated in figure 2.5. The meta-concept of `foo` is a specialisation of the default meta-concept, because it also maintains how many instances of `foo` are constructed. Using this architecture, we can change almost any aspect of the object-level computations by redefining that particular aspect in a meta-concept, so that the meta-concept will perform the appropriate computation when some object-level action is required

2.2 Knowledge-based systems

Rather than the general representation systems of the previous section, this section will discuss a number of specific application systems that use an explicit representation of control knowledge. The systems described in this section are of quite a diverse nature: NEOMYCIN

is a medical expert system; PRESS is an algebraic problem solver; GOLUX is a theorem prover; and PDP-0 models human problem solving behaviour. This diversity illustrates the validity of the assumptions discussed in the previous chapter 1 (separation of domain knowledge from control knowledge, explicit representation of control knowledge, and the use of a meta-level architecture for this purpose) across a wide spectrum of applications.

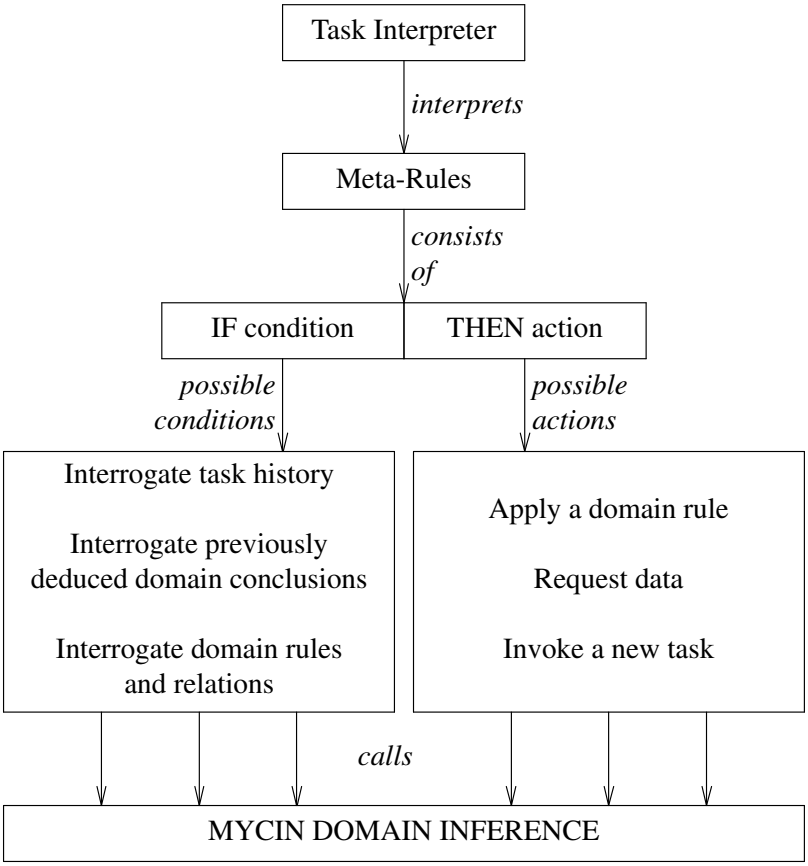


Figure 2.6: NEOMYCIN architecture

2.2.1 NEOMYCIN

NEOMYCIN, [Clancey and Bock, 1982], [Clancey and Letsinger, 1981], [Clancey, 1983b] is an expert system that diagnoses blood infection. It was created by making changes and additions to MYCIN [Shortliffe, 1976] to achieve an explicit representation of the control knowledge that was implicitly embedded in MYCIN. It implements a so called task interpreter on top of the MYCIN domain inference system to explicitly reason about the control of a session.

The architecture of NEOMYCIN is shown in figure 2.6. The main loop of NEOMYCIN is the task interpreter, which executes tasks. A task is a sequence of meta-rules describing how to achieve a particular task in the domain. During a task the meta-rules in that

task will be executed. From the current state of the session and based on the presence of particular domain rules the task interpreter decides whether to apply a domain rule, to request data or to call another task. Tasks are like conventional subroutines in that they can call each other, resulting in a stack-based scheduling of tasks. The basic elements of a NEOMYCIN task are:

- The focus: this is the argument with which the task is called. This often represents the object to which the task is applied (e.g. the focus of a task that identifies the nature of a biological culture would be the value of the variable `current-culture`).
- Three sets of meta-rules: the DOBEFORE, DODURING and DOAFTER rules that represent the prologue of the task, the main body and the epilogue respectively.
- The goal that is recorded to show that the task has been accomplished. This can be seen as the result of the task if it exits successfully.
- The end-condition that may abort the task when it becomes true. Whether the task is aborted or not depends on the task-type.
- The task-type, which specifies how the DODURING rules are to be applied. There are two dimensions to the task type: simple vs. iterative, and try-all vs. not-try-all. The combinations give four ways of applying the DODURING rules:
  1. Simple, try-all: The rules are applied once each, in order. Each time a metarule succeeds, the end condition is tested.
  2. Simple, not-try-all: The rules are applied in sequence until one succeeds, then the process stops.
  3. Iterative, try-all: All the rules are applied in sequence. If there are one or more successes, the process is started over. The process stops when all the rules in the sequence fail. Each time a metarule succeeds, the end condition is tested.
  4. Iterative, not-try-all: Same as for iterative try-all, except that the process is restarted after a single metarule succeeds.

The task concept of NEOMYCIN is very similar to the control blocks in S1. The representation language for tasks is rather different from the language for control blocks, but both control blocks and task implement a form of top-down control, where the domain task is (possibly recursively) divided into smaller sub-tasks by the control mechanism. However, once the sub-tasks request the solution of a particular domain problem, there is no longer any explicit control, until a solution is returned, after which the control layer takes over again.

2.2.2 PRESS

PRESS (**PR**olog **E**quation **S**olving **S**ystem) [Bundy and Welham, 1981], [Sterling et al., 1982], [Silver, 1986] and [Bundy and Sterling, 1988] is a system for solving algebraic problems, and manages to do so up to A-level[1]. One of the main features of PRESS is that the

[1]The exams qualifying for University taken by 18 year olds in England and Wales.

system proceeds in its problem solving process by trying to prove theorems at the meta-level, producing object-level proofs (i.e. solution of algebraic problems) as a side effect. The key idea of this methodology, called *meta-level inference* as described in [Sterling, 1984], is that strategies are considered to be at the meta-level of the domain. That is, the strategies are axioms of a meta-theory. This is of course entirely consistent with our definition of control knowledge as meta-knowledge in section 1.2.3. For example, consider the following meta-level axiom from PRESS:

```
singleocc(X, L=R) &
position(X, L, P)   &
isolate(P, L=R, Ans)
-> solve(L=R, X, Ans).
```

The declarative meaning of this meta-level axiom is:

> "If an object-level problem `L=R` contains exactly one occurrence of `X`, and the position of this occurrence in `L` is `P`, and if the result of isolating `X` in `L=R` is `Ans`, then `Ans` is a solution in `X` to the equation `L=R`, with `X` as the unknown."

However, it also has a procedural meaning:

> "In order to solve an equation `L=R` in `X`, check that `X` occurs exactly once in `L=R`, determine the position `P` of `X` in `L=R`, and isolate `X` in `L=R`, resulting in the answer `Ans`."

Note that this description refers to properties such as position and number of occurrences of `X`. These are meta-theoretic syntactic features. A meta-level axiom as the above would get used by a typical call to the `solve`-procedure of PRESS, as in for example:

```
?- solve(log(e, x+a)+log(e, x-a), x, Ans).
```

The inference of PRESS occurs at the meta-level. Some of the meta-level predicates (as indeed the example above) are of the form

`New` is the result of applying `Rule` to `Old`.

To satisfy such a predicate, the rule `Rule` is applied to the expression `Old` to produce `New`. As a result of executing such a meta-level predicate, an algebraic transformation has occurred at the object-level: the expression `Old` has been transformed into `New`. Thus, the object-level transformations are executed by performing inferences in the meta-theory. In fact, in PRESS the object-level theory (i.e. a theory of algebraic rewrite rules) does not have a separate existence. Rather, they are encoded within the meta-level theory as rule schemata such as:

```
isolation_rule(V, log(U,V)=W => V=U**W).
```

which would be used by the isolation procedure. PRESS achieves control of the object-level problem solving by simulating this problem-solving through the execution of the meta-level code. In other words, search at the object-level is replaced by search at the meta-level.

This works well because, as described in [Silver, 1986], the meta-level search space is much better behaved than the object-level space. In particular, the branching rate of the meta-level space is much lower, and most wrong choices lead to dead ends rapidly. The use of meta-level inference moves the search process from the object-level to the meta-level, and thereby transforms an ill-behaved search space to a better behaved one. If the meta-level space is still too complex, it is in principle possible to axiomatise the control of this level, i.e. to produce a meta-meta-level. This process can in theory be continued until the control process of the highest level becomes trivial. This usually happens very early. Only two levels are needed in the case of PRESS.

Related systems that have been constructed using a methodology similar to PRESS are MECHO [Bundy et al., 1979], IMPRESS [Sterling, 1982] and LP [Silver, 1986].

The problem of choosing between the various strategies that are axiomatised at the meta-level and that are applicable at any point in the problem solving process now depends on the proof procedure that is used for the meta-level interpreter. PRESS relies on the fixed behaviour of an implicit interpreter for the meta-level (i.e. the meta-meta-level interpreter) called the heuristic waterfall, which tries all meta-level strategies in a fixed order, applies the first strategy that is applicable, and starts again at the top of the list. The system described in [Takewaki et al., 1985] is a re-implementation of PRESS. It does allow reasoning about the selection of strategies (and thus provides a meta-meta-interpreter), rather than using a hardwired interpreter at the meta-level.

2.2.3 PDP-0

PDP-0 [Jansweijer et al., 1986] is designed to be a cognitive model of human problem solving behaviour, rather than just a high performance reasoning system, such as NEOMYCIN or PRESS, which don't claim any psychological validity. It models the behaviour of inexperienced human problem solvers in the domain of thermodynamics. Not only does the program model the domain reasoning done by a novice problem solver, but it also models the reasoning strategies employed by human problem solvers. For this purpose, the program contains explicit knowledge of problem solving strategies, and an explicit representation of its own behaviour. Furthermore, whenever difficulties arise in the object-level reasoning, it can make its own behaviour the subject of a diagnose-repair process that analyses the reasons for the problem that occurred in the object-level reasoning, and suggests a possible repair for the impasse, using strategical knowledge about problem solving techniques. The program's problem solving behaviour is driven by knowledge about the domain, consisting of knowledge about domain objects and the relations between them, and knowledge about domain independent problem solving strategies, in the form of production rules. Using a goal-driven approach, the system selects a problem-solving strategy from a library of general strategies. This choice is determined by properties of the current goal. The application of strategies on goals generates a goal-tree, which is explicitly represented, annotated by a trace of problem solving actions.

When the program comes to a dead end, for example because none of the known strategies is applicable, or because of the unexpected failure of an applied strategy, this will be noticed by a supervising component. The supervisor will then ask a meta-problem solver to propose an adjustment of the current goal-tree, in order to solve the impasse.

The meta-problem solver tries to solve the difficulties of the object-level problem solver in a three step process. First the difficulty is categorised into one of a few types of impasses. After this it uses general heuristics to propose repairs for each class of impasse. The final step is the specification of the general repair plan into a concrete new or adjusted goal-tree.

This approach is similar to that taken in VMT [Hudlicka and Lesser, 1984], where the meta-level part of the expert system monitors the object-level of the system. If the latter performs badly, according to some explicitly stated criterion, the former applies fault finding techniques to "diagnose" the object-level. This diagnosis is then applied to adjust the techniques used by the object-level.

2.3 Theorem provers

The systems described in the previous section were built to apply meta-level inference techniques to knowledge-based systems. The systems that we will describe in this section are not knowledge-based systems, but rather general purpose theorem provers which perform general, domain independent logical reasoning. Again, the property that all these systems have in common is the explicit representation of control knowledge, and the use of a meta-level architecture for this purpose.

2.3.1 GOLUX

GOLUX is a theorem prover written in the early '70s [Hayes, 1973], [Hayes, 1974]. Theorem provers obviously suffer from the problem of controlling their inference process, since at any time in a proof, many possible inference rules can be applied (as in a natural deduction system), or one inference rule can be applied to many different formulae (as in a resolution system). The use of an explicit meta-level to program a specialised search strategy is a central theme in GOLUX. In GOLUX, control over the inference process is obtained by describing the desired behaviour of the system, i.e. by making assertions about the desired behaviour. When the system is given such an assertion, it must "obey" it (if possible), i.e., behave in such a way that the assertion is made true. The behaviour of the system must always be consistent with the control assertions in force at a given moment. In general, a control assertion is of the form:

$$(Q_1\ S\ Q_2\ P)[p(P,S)]$$

where Q_1 and Q_2 are quantifiers, P ranges over possible proofs, S ranges over possible states of the theorem prover, and p is a predicate on the syntactic form of proofs. An example is:

$$(\forall S)[\neg(\exists P_1, P_2)[r(P_1, P_2, S) \wedge P_1 \neq P_2]]^2$$

If the predicate $r(P_1, P_2, S)$ says that in state S the subproofs P_1 and P_2 descend from a common parent, the above assertion can be used to insist that a proof has at most one descendant. This would make the computation deterministic.

[2] Although Hayes gives this formula as an example of a control assertion, it does not actually reflect the general form of a control assertion as specified by him. However, it can be easily reformulated in the required form by putting it into prenex normal form.

To allow the formulation of the above expression we need to be able to talk about proofs and states. Hayes defines states to consist (conceptually) of a set of active assertions (in both object- and meta-language), and a set of (partially completed) proofs. Proofs are uniformly represented as trees of formulae. The control language offers a quoting mechanism to allow the formulation of predicates on the syntactic form of object-level formulae. Finally, the inference rules that the system uses, are explicitly represented as operators. This allows expressions like

$$R_1(S_1, next(S_1))$$

which says that operator R_1 can be applied in state S_1 to obtain the successor state of S_1, denoted by $next(S_1)$.

Problems arise of course with the mechanism that should enforce these control assertions on the theorem-prover. One of these is that the collection of control assertions active at a given moment does not completely define the behaviour of the interpreter. GOLUX programs are thus non-deterministic, and Hayes does not specify what GOLUX does when its behaviour is underspecified. Unfortunately, the implementational details of how the control assertions of GOLUX influence the behaviour of the theorem prover were never published.[3]

2.3.2 NuPRL

The NuPRL system [Constable et al., 1986] was developed at Cornell University as an interactive environment for creating proofs in a formal theory of constructive mathematics. The object-level language which represents these formal theories is Martin-Löf's Intuitionistic Type Theory [Martin-Löf, 1973,Martin-Löf, 1982]. An interesting feature of the NuPRL system is that the logic takes account of the computational meaning of proofs. For instance, given a constructive existence proof, the system can use the computational information in the proof to build a representation of the object which demonstrated the truth of the existential assertion. As an example, consider the formula:

$$\forall \text{ l:int list } \exists \text{ l':int list } [\forall \text{ x:int } [\text{member(x, l)} \leftrightarrow \text{member(x, l')}] \wedge \text{sorted(l')}]$$

which states that for every list of integers l there is another list of integers l' with the same members as l such that l' is sorted. A proof of this formula will give rise to an algorithm for actually sorting a given list l_0 into a sorted list l'_0. Using this feature, the NuPRL system can be used for the generation of programs from specifications.

The NuPRL system is interactive, and requires the user to specify which object-level rules of inference should be used at any point in a proof. However, the system provides a functional language for combining elementary rules of inference into *tactics*: combinations of a (possibly large) number of rules of inference into a single step. The steps can be combined using a number of operators called *tacticals*, as listed in figure 2.7. More precisely, these tacticals do not combine rules of inference, but they combine tactics, so that

[3]Personal communications revealed that GOLUX was implemented using a generate-and-test cycle: it first generated all possible inferences from a given formula and then proceeded to remove all those inferences that violated any of the control assertions. The behaviour of GOLUX with underspecified control assertions remains unclear.

Tactic	Description
`R1 then R2`	apply `R1` to the current goal, and, if successful, apply `R2` to all the resulting subgoals.
`R1 or R2`	apply `R1`, or, if `R1` failed to apply, apply `R2` to the current goal.
`repeat R`	apply `R` to the current goal, and recursively apply repeat R to the resulting subgoals until `R` no longer applies.
`try R`	apply `R` to the current goal if possible, but do not fail if R is not applicable
`complete R`	apply `R` to the current goal but only succeed if no subgoals remain.

Figure 2.7: some NuPRL tacticals

Function	Type	Description
destruct-conj	term→term × term	destructs a conjunction into two conjuncts
term-kind	term→token	reports the kind of a logical term (such asatom, predicate, conjunction, etc.)
goal	term	returns the current goal
hyp	int→term	returns the n-th hypothesis in the current proof
hypotheses	term list	returns the list of all hypotheses in the current proof

Figure 2.8: some NuPRL-ML functions

nested expressions such as `(repeat T1) then (try (T2 or T3))` are possible. To allow elementary rules of inference to be used, the system provides for every rule of inference R a corresponding tactic T_R which only applies that particular inference rule.

Not only can the user combine tactics using tacticals, but NuPRL also offers a general purpose programming language, ML [Gordon et al., 1979], in which arbitrary computations can be performed before deciding which tactic to apply to a goal. For this purpose, ML is extended with a large number of extra functions that can be used to inspect the formulae occurring in the current proof. Figure 2.8 lists some of these functions as examples. The notions of tactics and tacticals, and the use of ML as a meta-level language was inherited from the Edinburgh LCF system [Gordon et al., 1979] on which NuPRL is based.

Because it is possible to write arbitrarily complex ML programs that can inspect the current proof and then decide which rules of inference (or more precisely, which tactics) to apply, these tactics can be seen as meta-level programs which reason about the control of the object-level inference rules. A feature worth mentioning is the way in which the system ensures the soundness of any tactics written by the user. All functions in ML

(either built-in or user-defined) are typed: the types of their arguments and their value is either declared or can be inferred from other declarations. The version of ML used in NuPRL (again based on ideas from the Edinburgh LCF system) has a special type called *proof*, and all built-in ML tactics (known to be correct) are of type *proof→proof*[4]. In particular, all tactics corresponding to object-level rules of inference are of this type. Furthermore, all predefined tacticals (such as the ones listed in figure 2.7) are of type *(proof→proof)* → (proof→proof)*, in other words, they map a number of tactics onto a new tactic. Now, a user defined tactic is only considered to be syntactically correct if the system is able to infer that it is indeed of type *proof→proof*, in other words that it maps a correct partial proof onto a correct partial proof. In this way, it is impossible for a user to write tactics that produce unsound proofs.

2.3.3 Proof plans

A second effort to automate the construction of proofs in Intuitionistic Type Theory is based on the notion of proof plans [Bundy, 1988], [Bundy et al., 1988]. The essential idea behind proof plans is to not use meta-level programs like ML-tactics in the search for a proof, but instead to write specifications of these meta-level programs, and to then use these specifications in the search for a proof, rather than the tactics themselves. These specifications are called *methods*, and are expressed in a meta-logical language containing primitives to inspect and construct logical formulae occurring in the object-level proof. A method consists of a number of slots, the most important of which are:

- An input-slot, specifying the form of the object-level formula to which the method is applicable.
- A precondition-slot, specifying further conditions that must be true for the method to be applicable.
- An output-slot, specifying the form of the object-level formulae that will be produced as subgoals when the method has applied successfully.
- A postcondition-slot, specifying further conditions that will be true after the method has applied successfully.
- A tactic-slot, specifying the name of the tactic for which this method is a specification.

An example of a particular method is the `unfold` method shown in figure 2.9. This method rewrites one occurrence of a recursively defined term using the step equation for that term. The input-slot of the method (2), can be any sequent `H⊢G`, where `H` is the hypothesis and `G` is the goal. The argument `[N|Pos]`, to the name, `unfold`, of the method (1), and the tactic (6) is a list of numbers specifying a position of the term to be rewritten in the goal `G`. The preconditions, (3), for attempting the tactic are as follows. In position `[N|Pos]` in `G` there should be a constructor term, `Constructor` with a constructor function `ConstructorFunc` as its dominant function, `F`, whose recursive definition has the step case

[4]In fact, tactics map partial proofs onto partial proofs, so that the name of the type *proof* is somewhat misleading.

```
method(unfold([N|Pos]),                                    (1)
       H |- G,                                             (2)
       [type(_, _, _, Constructor),
        exp_at(Constructor, [0], Constructor_Func),
        exp_at(G, [0,N|Pos], Constructor_Func),
        exp_at(G, [0|Pos], F),
        prim_rec(F, N),
        step(F, StepEq)
       ],                                                  (3)
       [rewrite(Pos, StepEq, G, NewG)],                    (4)
       [H |- NewG],                                        (5)
       unfold([N|Pos])                                     (6)
      ).
```

Figure 2.9: an example method

`StepEq`. The result of a successful application of the tactic will be that the output, (5), will be a sequent `H`⊢`NewG`, in which `NewG` is formed from `G` by rewriting the term at position `Pos` using `StepEq` (4).

These methods can be regarded as operators in the sense of plan formation: they transform one state of the proof (as specified by the input-slot) into another state of the proof (as specified by the output-slot). These methods can then be fitted together to form a sequence of methods such that each method in the sequence is applicable to the output formula of the previous one, with the final method having no more subgoals to prove. Standard planning techniques can be used to construct these sequences, which are called *proof plans*. Advantages of using the methods in the search for a proof rather than the corresponding tactics are firstly that the methods are written in a meta-linguistic logic, and are thus more intelligible than the tactics written in ML, and secondly that the computation of the pre- and postconditions of the methods is more efficient than the execution of the tactics.

A further considerable advantage of the use of methods combined with planning techniques instead of tactics is that the methods are not required to be full specifications of the (provably sound) tactics. Instead, they can be partial specifications which only compute part of the required pre- and postconditions for a method. The gain of this is that the power (and therefore the cost) of the pre- and postconditions of the methods can be adjusted at will to balance applicability and specificity. The price of this is of course that the plans which are built out of these method are then no longer guaranteed to succeed when it comes to executing the corresponding combinations of tactics. This may lead to having to replan part of the original sequence of methods and retrying the execution.

This approach of building proof plans before executing them is very different from the other systems described in this section, all of which combine the use of meta-level control knowledge with the inference in the object-level theory, rather than separating them as in

proof planning and execution.

2.4 Programming languages

In this final section of the review of meta-level systems in the literature, we will discuss a few programming languages that allow the explicit representation of control information. Of course, the knowledge representation languages discussed in section 2.1 are also strictly speaking programming languages, but they are designed for a very specific purpose, whereas the languages we describe here are general purpose programming languages. We will describe two Prolog based systems, and one system based on LISP.

First we will discuss a small example in Prolog which shows that the explicit representation of control knowledge is not only useful in reasoning systems, as discussed in chapter 1, but also in the context of a general purpose programming language such as Prolog. Prolog is based on resolution over a set of Horn Clauses in first order predicate calculus. Although the resolution procedure uniquely determines a set of answers for a given query, it does not determine the order in which these answers will be produced, and thus some control regime needs to be enforced on the resolution procedure. Standard Prolog systems impose a hardwired "top-down, left-to-right" computation rule, in which clauses are selected in a fixed top-down order, and conjunctions are resolved left-to-right. Such a uniform and fixed computation rule is unsatisfactory. As an example, consider the Prolog clause:

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

together with a (potentially large) set of ground clauses specifying `father`-relations. If we use this clause on a query like

```
:- grandfather(pete, Y).
```

the left-to-right computation rule is quite sufficient, because it first solves the conjunct `father(pete, Z)`, which presumably only results in a small number of possible bindings for `Z`, each of which can be tried in the fixed top-down order in the second conjunct. However, if we consider the goal

```
:- grandfather(X, pete).
```

the fixed left-to-right computation rule does not work so well. It will first compute `father(X, Z)`, resulting in many possible bindings for `X`, each of which has to be tried in the second conjunct, where almost all of them will fail. In this case it would have been better to select the second conjunct first, although either choice will eventually result in the same set of bindings for `X`. The only[5] solution for this problem in a standard Prolog system is to write two specialised versions of the original clause:

```
grandfather(X, Y) :- var(Y), !, father(X, Z), father(Z, Y).
grandfather(X, Y) :- father(Z, Y), father(X, Z).
```

using the non-logical features of Prolog, such as `var/1` and the cut, and mixing the control of the computation with the logical contents of the computation.

[5]Other Prolog systems, such as IC-prolog, [Clark and McCabe, 1982] solve this problem by annotating the object-level variables with control information. Since such systems do not have an explicit meta-level architecture, we will not discuss them here.

2.4.1 Gallaire/Lasserre

The Prolog system described in [Gallaire and Lasserre, 1982] is an attempt to solve problems like the one sketched above (to escape from the fixed computation rule without mixing logical contents and control) by providing an explicit and separate way to represent the desired control of the Prolog computation, using meta-level Horn clauses. Meta-predicates can be defined that handle both clause selection and conjunct ordering. These predicates are then used in an interpreter-loop to determine the behaviour of the system. A fixed vocabulary of meta-predicates is available that can express a variety of properties of the object-level propositions that compete for execution. These properties include among others the number of literals in a clause, the presence of a particular literal in a clause, the value of any ancestor of a clause, and the invocation depth of the clause. The object-level propositions that are present in the knowledge base can be specified in the meta-rules by their position in the knowledge base, or by a (partial) specification of their contents. In the Prolog syntax that is used in the system, a clause like

```
order(p(X,Y), [N1, N2, N3, ..., Ni]) :-
        C1, C2, C3, ..., Ck.
```

states that for the resolution of literals that are an instantiation of `p(X,Y)` the clauses numbered `N1, N2, ..., Ni` will be used in that order, provided the conditions `C1, ..., Ck` are met. Notice that the variables `X, Y, N1, N2, ..., Ni` can be used in the conditions `Ci`. An example of this would be

```
order(p(X), [1, 2, 3]) :- cond1(X).
order(p(X), [1, 3, 2]) :- cond2(X).
order(p(X), [3, 1, 2]) :- cond3(X).
```

This specifies a different order for the clauses for `p(X)` for different conditions on the argument `X`.

An example of content directed conflict resolution is

```
before(p(_), Clause1, Clause2) :-
        length(Clause1, _, N), length(Clause2, _, M),
        N < M.
```

which states that for the resolution of literals of the form `p(_)` shorter clauses will be used before longer clauses. Replacing `p(_)` with either a variable or a more specific term (e.g. `p([])`) would enlarge or reduce the scope of this heuristic. Both position directed and content directed conflict resolution allow the formulation of domain dependent and domain independent strategies.

The system also provides a mechanism for conjunct ordering. A clause of the form

```
need(p(X, Y, Z)) :- inst(X).
```

says that for the selection of literal `p(X, Y, Z)` for execution, it is necessary that variable `X` has been instantiated. This possibility for conjunct-ordering can be used to solve the problem with the `grandfather` example described above. Adding the control instruction:

```
need(father(X, Y)) :- inst(X) ; inst(Y).
```

would cause the interpreter to always prefer partially instantiated calls to `father/1` over uninstantiated calls, and would thus optimally execute both the query

```
?- grandfather(pete, X).
```

and the query

```
?- grandfather(X, pete).
```

using the single clause

```
grandfather(X, Y) :- father(X, Z), father(Z, X).
```

A literal can be mentioned directly (as the literal `p` in the example above), or it can be designated indirectly, using an expression like:

```
literal(X, Name, ListOfProperties)
```

This indicates a literal `X` named by Name that satisfies each of the properties on `ListOfProperties`. This is a list of pairs `(Pi:Vi)`, where `Pi` is the name of a property, and `Vi` is its required value. The properties that are available include such things as `ancestor`, `father`, `depth` and `solved`. For instance, a meta-level clause like

```
before(T1, T2) :-
        literal(T1, X, [depth:N1]),
        literal(T2, Y, [depth:N2]),
        N1 < N2.
```

specifies a breadth-first strategy for the interpreter. Further restrictions on `X` and `Y` would impose such a strategy only on the named literals.

In earlier work along the same lines [Gallaire and Lasserre, 1979], further facilities were proposed to

- assign priority numbers to competing clauses,
- block backtracking over specified clauses (corresponding to dynamic cut introduction)
- inhibit the execution of literals until they reach some degree of instantiation.

These declarative meta-rules are all used in the interpretation process, which is a literal-selection - clause-selection loop. The basic loop of the interpreter looks like:

```
goal(G) :-
        select-literal(G, L),
        select-clause(L, C),
        substitute(G, L, C, R1),
        goal(R1).
```

The meta-predicates are used in the first two stages of this control loop to select literals and clauses. However, the definition of this loop is a fixed part of the system and is not available for modification by the user. Thus, this central part of the control knowledge is not fully explicitly represented in this system (since our definition of an explicit representation, section 1.2.2, required both inspectability and modifiability). As a result, the system only allows the construction of chronologically backtracking, backward-chaining Horn clause interpreters. The behaviour of the system can only be affected by redefining the selection strategies for literals and clauses, using the primitives made available for this purpose. Other dimensions of the system's search strategy, such as its backtracking behaviour, cannot be influenced at all. Another problem with the system is that there is no clear distinction between meta-level and object-level language. Prolog is used for both languages (and also as the implementation language of the system). Although this mixing of levels allows for an efficient implementation, it can give rise to serious confusion, both at a conceptual design level and in the implementation.

An essentially similar approach is found in [Devanbu et al., 1986]. They also add control rules for Prolog programs. These control rules specify appropriate behaviour of the Prolog clauses on the basis of the standard 4-port execution model for Prolog. Another example of this approach is the METALOG system [Dincbas and Le Pape, 1984]. Although the METALOG system also provides the use of control-knowledge at other points in the control cycle (such as during backtracking) the main features from METALOG resemble the system described above. Yet another similar, though somewhat extended approach can be found in [Eshghi, 1986]. This system not only allows literal and clause selection, but also what Eshghi calls node selection: the ability to expand nodes in different parts of the proof tree. Furthermore, Eshghi provides an explicit representation of the object-level proof tree (unlike Gallaire and Lasserre and Devanbu et al.). Another system based on this approach is RLOG [Kramer, 1984].

2.4.2 Bowen/Kowalski

A different approach to the same problem (the explicit representation of control in Prolog) is taken in [Bowen and Kowalski, 1982]. The main feature of their system is that it explicitly represents the provability relation of the object-level language (Horn clause logic). This provability relation, in logic normally written as $P \vdash G$ (G is provable from P) can be formalised in Prolog as the `demo/2` predicate:

```
demo(Prog, Goals) :- empty(Goals).
demo(Prog, Goals) :-
        select(Goals, Goal, Rest),
        member(Proc, Prog),
        rename(Proc, Goals, VariantProc),
        parts(VariantProc, Concl, Conds),
        match(Conc, Goal, Subst),
        apply(Subst, Conds+Rest, NewGoals),
        demo(Prog, NewGoals).
```

Notice that this formulation of provability requires the ability to explicitly refer to object-level programs (or theories, sets of clauses), something that is not required in the Gallaire/Lasserre system. Different control strategies can now be implemented by changing the above formulation of `demo` or its constituent predicates. For instance, clause selection or conjunct-ordering can be changed by redefining `member` or `select` respectively. Since `demo` formalises provability, this approach not only allows us to change the control of the object-level but also its logical properties. However, this is not our prime interest here, since we are mainly interested in the flexibility of the system with respect to its control. Thus, while the declarative reading of the above definition of the `demo` predicate defines the provability relation of the object-level, its procedural interpretation gives us a definition of the inference strategy. This approach of formalising the provability relation of the object-level language at the meta-level is of course very close to what PRESS does in the context of algebraic problem solving. Just as in PRESS, the system completely simulates the object-level problem solving at the meta-level, by replacing every object-level problem $A \vdash_L B$ by the corresponding meta-level problem $Pr \vdash_M demo(A, B)$, where L and M are the object-level and meta-level language, and $\vdash_L$ and $\vdash_M$ represent object-level and meta-level provability, and Pr is the representation of the `demo` predicate at the meta-level. However, Bowen and Kowalski argue that

> "... many object language problems can be solved more naturally and more efficiently in the object language than in the metalanguage. Thus it is desirable to combine the directness of the object language with the power of the metalanguage in an amalgamation which facilitates the communication of problems and their solutions between them."

Such communication can be accomplished by means of rules that link the object-level and the meta-level:

$$\frac{Pr \vdash_M demo(A', B')}{A \vdash_L B} \qquad\qquad \frac{A \vdash_L B}{Pr \vdash_M demo(A', B')}$$

The first rule allows the meta-level language to communicate the solutions of object-level problems to the object-level language, whereas the second rule allows the object-level language to communicate the solutions of its problems to the meta-level language. [Weyhrauch, 1981] calls these rules *reflection principles*, after [Feferman, 1962]. To complete the amalgamation of L and M, we need a *naming relation* which associates with every linguistic expression of L at least one variable-free term of M. In the above rules, A' and B' are the meta-level names in M of the object-level theory A and the object-level expression B. Bowen and Kowalski are particularly interested in the amalgamation where $L = M$, i.e. the two languages are identical. This case is of special interest for logical reasons, since it allows the formulation both of sentences which mix object-level language and meta-level language, and of self-referential sentences. From the viewpoint of control and of system-architecture in general, the case where $L = M$ also plays a special role, which will be discussed in later chapters.

The `demo` predicate described above, formalising the derivability relation of the object-level, is itself implemented by the very same Prolog processor that it models. Such a construction (where an expression in some computational formalism models its own computational process), is known as a *meta-circular interpreter*. "Meta-" because it models another level of computation, and "circular" because it does not provide a well-founded definition of the computational formalism: a meta-circular interpreter (from now on: MCI) has to be run by the very formalism that it models in order to yield any sort of behaviour. An MCI provides a two-level architecture, where the MCI defines explicitly (at the meta-level) the execution of code at the object-level, in such a way that this object-level code, when interpreted by the MCI behaves the same as when executed by the base implementation. However, we can of course drop this last constraint, and change the code of the MCI, and thereby the way it will execute object-level code, providing us with a system with an explicitly represented (i.e. both inspectable and modifiable) control regime at the meta-level. As stated, the code for such an MCI is itself executed by the base-level implementation, and as a result its interpretation process (i.e. the meta-meta-level interpreter) is implicit in the system and can neither be inspected nor modified, restricting us to a two level system. 3-LISP is an attempt to generalise this situation to an n-level system, for arbitrary n.

2.4.3 3-LISP

3-LISP is meant as a general purpose function programming language with a meta-level mechanism that allows the user to modify the underlying interpreter, for instance how it deals with evaluation order, variable bindings, function definitions, debugging information, etc. A program in 3-LISP [Smith, 1984], [des Rivières and Smith, 1984] is executed by an MCI for 3-LISP (i.e. written in 3-LISP itself). However, unlike the MCI used by Bowen and Kowalski in their Prolog system, the MCI in 3-LISP is itself again interpreted by another MCI etc., ad infinitum. This gives full inspectability at all levels of interpretation of the system, but not yet modifiability. In order to achieve this, functions in a 3-LISP program can be declared as "reflective"[6], which means that during their execution the system will "reflect up", moving up one level in the infinite tower of MCIs. If a function `f` is declared as a reflective function, and the expression `(f e1 ... en)` is encountered by an interpreter at level n, then the function body of `f` is called with three arguments: the original argument list `(e1 ... en)`, the current variable binding environment and the current continuation[7], and this call will be executed by the interpreter at level $n+1$.
The binding environment and the continuation together completely determine the state of the interpreter at level n, and since the function `f` now has access to these two values, it can itself determine the course of the computation at level n. Notice that the binding environment plus the continuation determine the flow of control in an MCI, but many other things are left implicit: how errors are processed, how data structures are implemented,

[6]This terminology is chosen because a program at level n in the tower can "reflect" upon the behaviour of the program at level $n-1$.

[7]See [Steele and Sussman, 1978] for a detailed description of continuation-passing interpreters. For our purposes it is enough to say that a continuation consists of all the code to be executed by the interpreter after the current function exits.

```
(define REFLECT boundp (symbol &optional env cont)
    (funcall cont
        (if (mapcan '(lambda (binding) (eq (car binding) symbol)))
            env
            (cons (cons symbol nil) env))))

(define SIMPLE foo (...)
    ...
    (if (boundp bar) ...)
    ...)
```

Figure 2.10: a reflective function in 3-LISP

how I/O is carried out, etc. All these things are buried in the primitive procedures of 3-LISP and are made neither inspectable nor modifiable: the reflective capabilities of 3-LISP only concern the flow of control. An MCI can be viewed as an account of how a language is processed, and as such it explains various things about how the language is processed, but many other things are not explained.

As an example of the use of the reflective facility, we can define a function to be executed at level $n+1$ which checks if a certain variable is bound at level n and provides a default binding `nil` if necessary, as shown in figure 2.10[8]. If the function `foo` is executed at level n, then the reflective function `boundp` will be executed at level $n+1$ with the environment and the continuation of level n as extra arguments, allowing it to inspect and modify the execution of the computation at level n.

One question that has to be settled for an implementation of 3-LISP to be possible is the obvious threat of the infinite regression of MCIs, each MCI_n being interpreted by MCI_{n+1} without any base implementation at the top to close the tower. The following argument, as given in [des Rivières and Smith, 1984] shows that although the user can think of 3-LISP as an infinite tower of MCIs, in practice only a finite number of levels is needed. The key observation is that the activity at most levels - in fact at all but a finite number of the lowest levels - will be monotonous: the MCI will primarily be used to process the same expressions, namely those that make up the MCI itself (or more precisely, those that make up the MCI of the level below). From some finite level k all the way up, the tower will just consist of MCIs interpreting an MCI. Smith and

des Rivières call a processing level "boring" if the only code that is processed at that level in the course of a computation is the code of the MCI of the level below. The degree of reflection Δ of a user-program is defined as the lowest n such that when the program is run at level 0, all levels higher than n are boring. Thus, user programs that do not use any reflective capabilities have $\Delta = 1$. 3-LISP only requires programs with a finite

[8]The syntax in this figure is not exactly according to the original 3-LISP specification, given in [des Rivières and Smith, 1984]. Instead, to improve readability we have adopted a CommonLisp-like notation, as in [Maes, 1987], from which this example is taken.

degree of reflection to terminate, just as a correct implementation of recursion is only required to terminate if the recursion depth is finite. We can assume the existence of an implementation G (G for ground) that executes programs with $\Delta = 1$ (after all, such a G only has to interpret 3-LISP programs stripped of all reflective capabilities). Given G, the possibility of an implementation for programs of $\Delta = n$ for any n follows by an obvious inductive argument on n. In fact, this is more or less how the actual implementation of 3-LISP described in [des Rivières and Smith, 1984] works. Notice that this implementation is very similar to the implementation of KRS described above, where the reflective process (in KRS realised by defined meta-concepts) can go up arbitrary many levels, until the first level that does not have a meta-concept is executed by the base implementation.

Chapter 3

Analysis of the literature

In the previous chapter we discussed a number of systems, built for different purposes, that all represented different solutions to the problem of explicitly representing control knowledge. Because of this diversity, it seems very difficult to compare and classify these systems and the solutions they offer, in order to discuss which solutions are good ones and which are not. Exactly such a comparison is the task of this chapter.

The most obvious distinction that can be made between the systems presented in the previous chapter is the language that is used to represent the domain knowledge in the system. This varied from Horn Clause logic to a functional language, and from an object-oriented language to production rules. Each of these particular choices of representation language brings with it its own version of the control problem (the problem of how to control the search space that is generated by the domain representation). For Horn Clause logic, the system has to decide (among other things) on conjunct-ordering and backtracking strategies, an object-oriented system has to decide how to handle messages, a production rule language has to do conflict resolution, etc. All these control problems are quite different from one another. Furthermore, it is well known that appropriate choices of both a representation language and an ontology to describe the domain problem can greatly reduce the size of the control problem for a specific application. Much of the research in knowledge representation is exactly about finding a suitable language (e.g. Horn Clause logic) and a suitable ontology (i.e. what particular predicates, functions and constants are chosen) that reduces the search space for a particular problem to a feasible size. A classic paper that illustrates this phenomenon in the context of the "Missionaries and Cannibals" problem is [Amarel, 1968]. In general, we can distinguish two aspects of dealing with the control problem for a particular application. First there are the choices of the representation language, an ontology for that language, and the corresponding inference engine. Second, we have to define a control regime for that inference engine in such a way that the inference engine behaves suitably for our particular application. Both choices affect the control problem. This is obvious for the second choice, but, as argued above, also true for the first. In general, it is impossible to say how much of the control problem that must be solved by formulating a control regime is dependent on the choice of the representation language and of the ontology. If the formulation of the control regime (which as we argued in the previous chapter should be done explicitly at the meta-level) was completely dependent on the choice of the object-level representation language and its ontology, then there

would be very little that we could say, in general, about meta-level architectures, without committing ourselves to a particular combination of object-level representation language and ontology. It is certainly true that for more detailed observations about meta-level systems we will have to commit ourselves to a particular representation language, and that is exactly what we will do later in this book (from the next chapter onwards). However, it is possible to abstract away from the particular representation language, and to look at the general architecture of the system, that is: to look only at the *relation* between object-level and meta-level, without looking at the formalisms and ontologies that determine their *contents*. Perhaps surprisingly, such an abstract point of view reveals that the diversity of systems discussed in the previous chapter can be classified into a limited number of typical architectures. The advantage of such a classification, based on essential architectural features rather than on properties incidental to a particular representation language, is that it will allow us to compare the different meta-level architectures independently from the choice of their representation language.

In this chapter we will first develop a classification based on our main distinguishing characteristic, leading to a classification in which all the systems from the previous chapter have a place. After that we will discuss a number of other, independent properties by which these systems can be distinguished, although they are of secondary importance. Again, we will classify each of the systems from the previous chapters with respect to these properties. Finally, we will compare all the different types of systems within these characteristics, and argue in favour of one particular type of system.

3.1 Classification of meta-level architectures

The essential characteristic of meta-level architectures is, of course, that they consist of two levels, the object-level and the meta-level. Each layer can be seen as an individual system with a representation language and an interpreter for expressions in that language. The system as a whole can at any moment be active at one of the two levels; either it is interpreting object-level expressions (using the object-level interpreter), or it is interpreting meta-level expressions (using the meta-level interpreter). This leads us to the notion of the *locus of action* (using a phrase coined in [Welham, 1986]): the place in the system which is active at any one point in time. This locus of action can then be either the object-level interpreter or the meta-level interpreter. It is exactly this locus of action that will form the basis for our main classification of meta-level architectures. We will distinguish a spectrum of systems, with at one end of the spectrum systems where the locus of action is almost all the time the object-level interpreter, and where meta-level activity takes places only occasionally. At the other end of the spectrum we will see systems where the converse is true: almost all (or sometimes: all) system activity takes place in the meta-level interpreter, and (almost) no activity takes place at the object-level. The systems in the middle of this spectrum exhibit equal amounts of object-level and meta-level activity. This classification, plus further subdivisions, is shown in figure 3.1, and we will discuss each of the types of system in this classification in the following subsections.

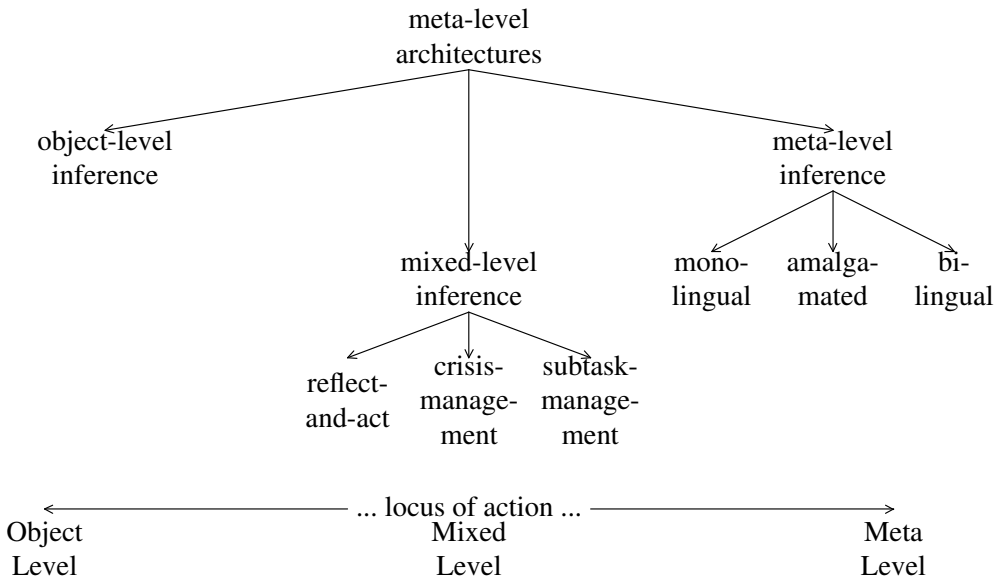


Figure 3.1: classification of meta-level systems

3.1.1 Object-level inference systems

On one extreme of the classification shown in figure 3.1 are the systems where the main activity is in the object-level interpreter. In fact, these systems do not have a separate meta-level interpreter (i.e. an interpreter for meta-level expressions), but only a built in (i.e. implicit, not inspectable and modifiable) object-level interpreter that takes the meta-level expressions into account during its computational cycle in order to adjust its behaviour. As a result, the object-level interpreter executes two types of instructions: firstly, the object-level expressions it is supposed to interpret, and secondly the meta-level expressions that affect its behaviour. Typically, the object-level interpreter performs a fixed computational cycle, and the meta-level expressions are concerned with certain fixed points within this cycle.

A good example of a system from this category is the Prolog system from Gallaire and Lasserre (section 2.4.1). The object-level Prolog interpreter goes through a fixed literal-selection clause-selection loop which can be affected in certain parts by meta-level instructions concerning the selection of clauses and literals. Thus, there is no separate meta-level interpreter to execute the meta-level expressions (remember that the main literal-selection-clause-selection loop is not modifiable and does not therefore constitute an explicit meta-level interpreter). In the case of the Gallaire/Lasserre system, the meta-level expressions can affect either the object-level interpreter in general, or only when it is executing specific object-level expressions, depending on whether the meta-level instructions mention specific object-level expressions or not.

A second example of this type of architecture is GOLUX (section 2.3.1), where the behaviour of the object-level theorem prover is constrained by meta-level assertions that it must "obey", i.e. the behaviour of the object-level interpreter must always be consistent

with the control assertions.

3.1.2 Mixed-level inference systems

In the middle of figure 3.1 we find systems where the computation takes place in both the meta- and the object-level interpreter. Object-level and meta-level computations are interleaved, and some mechanism is provided for switching between the two. We can further subdivide this category of systems in the middle of the spectrum on the basis of the criterion that is used for switching between object- and meta-level, as shown in figure 3.1.

3.1.2.1 Reflect-and-act systems

Sometimes the meta-level interpreter is called very frequently, before or after every object-level step. This organisation has been called a *reflect-and-act* loop, since the object-level interpreter "acts", the meta-level interpreter "reflects" on the actions of the object-level interpreter and these two together are chained together in a continuous loop. TEIRESIAS (section 2.1.1) and BB1 (section 2.1.3) are examples of such architectures

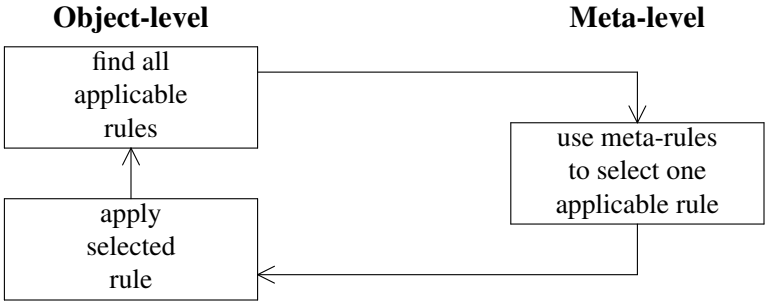


Figure 3.2: flow of control in reflect-and-act systems

The flow of control in such reflect-and-act systems can be described as in figure 3.2. The object-level interpreter finds the set of all applicable object-level rules and passes this *conflict-resolution set* on to the meta-level interpreter. For instance in TEIRESIAS, the meta-level interpreter uses its control knowledge to select one of these applicable rules, which is then handed down again to the object-level interpreter, which applies this rule. Similarly, in BB1, the object-level system finds the set of all active knowledge sources, and the meta-level system decides which of these knowledge sources should be applied, after which the object-level interpreter executes the selected knowledge source.

3.1.2.2 Crisis-management systems

Sometimes the meta-level is called only if a *crisis* or an impasse occurs in the object-level computation, for example when too many or not enough steps are possible at the object-level. PDP-0 (section 2.2.3) is an example of this approach.

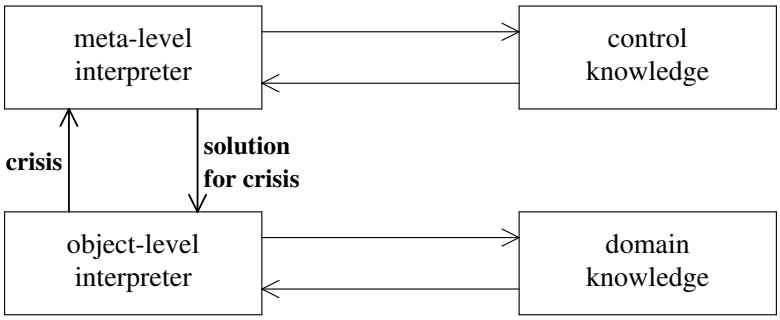


Figure 3.3: flow of control in crisis-management systems

The flow of control in crisis-management systems is summarised in figure 3.3. The object level interpreter uses the domain knowledge to solve a particular problem, and only hands over control to the meta-level interpreter if some kind of crisis occurs which prevents the object-level computation from continuing. The meta-level interpreter then uses its strategic (meta-level) knowledge base to try to solve this crisis. If some kind of solution has been found it is handed down to the object-level interpreter which can then proceed with the computation. Different kinds of crises can occur. One example of a crisis is when no object-level rules can be found that apply to the current subgoal. The meta-level interpreter then has to find a different subgoal for the continuation of the object-level computation. In this way we could implement user-directed backtracking, rather than the built-in standard behaviour of a system like Prolog. Another example of a crisis is when more than one object-level rule applies to the current subgoal. The object-level interpreter then turns to the meta-level for conflict resolution. In this way a reflect-and-act system can be simulated in an efficient way by a crisis-management system (efficient since the meta-level is only called if there is indeed more than one applicable object-level rule, rather than in every loop, as in reflect-and-act systems).

3.1.2.3 Subtask-management systems

Yet another approach is where the meta-level knowledge is used to partition the object-level task into a number of *subtasks.* In such a system, the meta-level interpreter decides on a task to be done, and this task will then be executed by the object-level interpreter. After completion of this object-level task (be it successful or not), the meta-level decides on the next subtask for the object-level. This approach is taken in S1 (section 2.1.2), NEOMYCIN (section 2.2.1), and MLA (section 2.1.4).

The flow of control in subtask-management systems is described in figure 3.4. The meta-level interpreter first decides upon a subtask to be solved by the object-level interpreter. The object-level interpreter then tries to solve this subtask, and only returns to the meta-level when it has either found a solution or when it has established that it cannot solve the subtask. On the basis of this result the meta-level interpreter can then try to find a new subtask to be solved. As can be seen from figures 3.3 and 3.4, the architectures for crisis- and subtask-management systems are very similar. The main differences are in

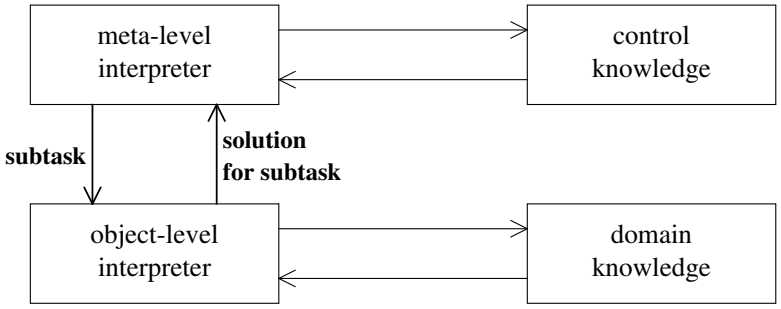


Figure 3.4: flow of control in subtask-management systems

the type of data that is passed between the object- and the meta-level (subtasks or crisis information), and in the place where the computation starts: crisis-management systems initiate their computations at the object-level, while subtask-management systems start their computation at the meta-level.

3.1.3 Meta-level inference systems

On the right side of the spectrum in figure 3.1 we see systems where the computation mainly takes place in the meta-level interpreter. In these systems the behaviour of the object-level is fully specified at the meta-level. Using this description of the object-level, the meta-level can completely simulate the object-level inference process. This means that there is no longer a need for an explicit object-level interpreter. As a result, the object-level interpreter is no longer present in the system, and its behaviour is completely simulated by the execution of its specification at the meta-level. Examples of this type of architecture are PRESS (section 2.2.2), 3-LISP (section 2.4.3), and the amalgamated logic by Bowen and Kowalski (section 2.4.2). For example in the amalgamated logic, the meta-level interpreter fully specifies the object-level computation, so that every aspect of it can be changed. The same holds for the other systems.

An important subdivision of meta-level inference systems can be made on the basis of the relation between the object-level language L and the meta-level language M used by the system. On the one hand, there are the systems that we will call *bilingual*. These systems support two strictly separate languages L and M. In order to provide upwards and downwards communication between L and M, the languages are related via a *naming relation* which translates sentences, sets of sentences, and other linguistic entities of L into variable free terms of M. The PRESS system provides such an approach, where the meta-level language is Prolog and the object-level language represents variables as designated Prolog atoms. Other bilingual systems discussed in chapter 2 are the NuPRL and the Proof Plan systems. On the other hand there are systems in which L and M are the same language. These systems can be divided into two subtypes. One subtype are the *monolingual systems*. In these systems , no syntactic distinction is made between object-level and meta-level expressions. Object-level and meta-level variables are both represented in the same way, and no account is given of the difference between object-level and meta-level

expressions. Examples of these systems are 3-LISP and the Gallaire/Lasserre system. For instance in the Gallaire/Lasserre system Prolog is used for both L and M. The other subtype of systems which have $L = M$ are the *amalgamated systems*. As in the mono-lingual systems, the amalgamated systems use the same language for both levels, but unlike the mono-lingual systems, the amalgamated systems do employ a naming relation as described above, so that each object-level expression has a variable free term as its name associated with it. (Since this variable free term is itself again a syntactically correct object-level expression, since $L{=}M$, it again must have another variable free term as its name, ad infinitum). An example of this approach is presented in the Bowen/Kowalski system described in section 2.4.2.

This concludes our classification of meta-level architectures according to their locus of action. This distinction corresponds roughly to a distinction made by Silver [Silver, 1986] between *object-level driven* and *meta-level driven* systems. Systems at the left end of the spectrum shown in figure 3.1 are object-level driven, since their main computation takes place at the object-level, and systems at the right end of the spectrum are meta-level driven.

3.2 Other properties of meta-level architectures

In this section we will discuss some more properties of meta-level architectures that can be used to further classify them, beyond the classification on the basis of the locus of action as described in the previous section. Some of the properties discussed in this section are found implicitly in the literature. The aim of this section is to sharpen these criteria, and to make them explicit. Figure 3.5 at the end of this section will summarise the position of all the systems described in the previous chapter with respect to all of these properties.

3.2.1 Linguistic relation between levels

The distinction between mono-lingual, bilingual and amalgamated systems used to subdivide meta-level inference systems in the previous section can in fact be used more generally, to apply also to object-level inference and mixed-level inference systems.

Object-level inference systems can have their meta-level instructions to the object-level interpreter expressed in a language that is either the same as or different from the object-level language. The Gallaire/Lasserre system relies on the two languages being the same, whereas GOLUX separates the two languages, providing an explicit *quoting mechanism* to give meta-level names to object-level expressions (using this mechanism, the ground terms at the meta-level that are used as the names of object-level expressions are always atomic constants).

The distinction between mono-lingual, bilingual and amalgamated systems also applies to mixed-level systems. TEIRESIAS is a bilingual system: although both object-level and meta-level language are production rule languages, they are quite different languages, properly separated. The languages are of the same type, namely production rule languages, but

they are not identical. This is similar to GOLUX, where both meta-level and object-level language are of the same type (first order predicate calculus), but are in fact separate languages. The same remark holds for MLA. NEOMYCIN and S1 are also bilingual systems, but in both these systems the meta-level language is not only different from the object-level language, but also of a different type. Both NEOMYCIN and S1 use production rules at the object-level, but use some special purpose language at the meta-level: the task-language in NEOMYCIN and the control blocks in S1.

3.2.2 Declarative or procedural meta-language

As already explained in section 1.2.1, the control knowledge of the meta-level can be either expressed in a *declarative* or a *procedural* language. Expressions in a procedural language can only be understood in terms of the *behaviour* of the meta-level interpreter, the actions it takes, the order in which it does things etc., whereas a declarative language states true facts that can be understood without reference to the behaviour of the meta-level interpreter. For example, the meta-level expressions of GOLUX are purely declarative descriptions[1]of properties of object-level proof trees, whereas something like the control knowledge of S1 is purely procedural in nature, talking about the order in which to perform actions, sequences and loops of instructions etc. Yet other systems (such as PRESS) have a meta-level language that has both a declarative and a procedural reading (as explained in the section on PRESS, section 2.2.2).

3.2.3 Partial specifications

An important property of some of the systems described in the previous chapter is that they allow the *partial specification* of meta-level knowledge. Such systems (like GOLUX, Gallaire/Lasserre, MLA, 3-LISP, KRS and Bowen/Kowalski) provide a *default specification* of the behaviour of the system that can be totally or partially overwritten by the user to modify the systems default behaviour. In some of these systems the default definition is available for inspection in the system (3-LISP, KRS, Bowen/Kowalski), whereas the other systems (GOLUX, Gallaire/Lasserre, MLA) only contain an implicit definition of their default behaviour.

3.2.4 Combinatorial completeness and soundness

As mentioned in chapter 1, one of the main purposes of having a meta-level architecture at all is to allow the object-level to be purely declarative, without having to worry about procedural aspects. Thus, for any given query, the object-level (implicitly) specifies a set of answers[2]. It is the task of the meta-level interpreter to determine which of these possible answers is going to be actually computed, and in which order. Furthermore, in

[1]It should be stressed that by "*purely declarative*" we do *not* mean that these expressions *only* have a declarative meaning, but instead that it is possible to give an account of the meaning of these expressions *without* any reference to a procedural interpretation.

[2]This is clearest if the object-level consists of a logical theory plus a set of logical inference rules, but similar notions exist for other declarative representation languages.

some systems, it is possible for the meta-level to produce answers not derivable from the object-level theory, using meta-theoretic devices like *reflection principles* [Feferman, 1962], implemented for instance in FOL [Weyhrauch, 1981]. We call a meta-level architecture *combinatorially complete* if it computes *all* the results derivable from the object-level theory (i.e. the meta-level does not *suppress* any object-level results). We call a meta-level system *combinatorially sound* if it computes *only* results derivable from the object-level theory (i.e. the meta-level does not *extend* the object-level results). Notice that it is possible for a meta-level architecture to be both *in*complete and *un*sound, namely if the meta-level suppresses some of the object-level results, but also computes extra ones.

With these definitions it is clear that completeness of a meta-level system is not always a desirable property. The whole point of a meta-level architecture is often to prune parts of the object-level search space, thereby suppressing certain object-level results because they are too expensive to compute, or unwanted in some other sense. [Wallen, 1983] used the term "positive heuristic" in connection with the concept of completeness: a positive heuristic is

> "a heuristic that prefers certain object-level computations over others, but that does not prohibit certain computations altogether."

In other words, a system that only allows positive heuristics is automatically complete. Unlike combinatorial completeness, combinatorial soundness of a meta-level system in the above sense is always a desirable property. In the context of using meta-level systems for control, we do not want to extend the results of the object-level theory. As described in section 1.2.3 one possible use of a meta-level architecture is to try and extend the results of the object-level theory, but this is not our interest here.

A system like the Gallaire/Lasserre Prolog system is one of the few systems described in the previous chapter that is complete: its only concern is the ordering of clauses and literals, thereby affecting only the order in which the object-level computation takes place, but not its ultimate outcome[3]. A system like GOLUX is certainly not combinatorially complete, as the example in section 2.3.1 shows: we can force proofs in GOLUX to be deterministic, thereby pruning alternative solutions. However, GOLUX is combinatorially sound in the above sense: no new object-level results can be introduced through meta-level computation. This is not true in meta-level inference systems like KRS and 3-LISP. Since their meta-levels completely specify the object-level computation, it is possible to extend the object-level behaviour in order to produce extra results.

This concludes our discussion of different properties to distinguish meta-level architectures. The table in figure 3.5 summarises the position of all the systems described in chapter 2 on these properties. This table also contains data on the Socrates system, which will be described in chapter 5. However, for convenience, we already include Socrates in the table here.

[3]Strictly speaking, this is only true for those versions of their system that do not allow dynamic cut-introduction. This corresponds to dynamically removing object-level backtrack-points, thereby potentially suppressing certain object-level results.

System	Architecture type	linguistic relation	declarative/ procedural	partial spec.	sound & complete
TEIRESIAS	reflect-act	biling.	decl.	yes	C– ; S+
S1	task-man.	biling.	proc.	?	C– ; S+
BB1	reflect-act	biling.	proc.	yes	C– ; S+
MLA	task-man.	biling.	decl.	yes	C– ; S+
KRS	meta-inf.	mono-ling.	_[1]	yes	C– ; S–
NEOMYCIN	task-man.	biling.	proc.	_[2]	C– ; S+
PRESS	meta-inf.	biling.	decl.&proc	_[2]	C– ; S–
GOLUX	object-inf.	biling.	decl.	yes	C– ; S+
PDP-0	crisis-man.	?	?	_[2]	?
Gallaire	object-inf.	mono-ling.	decl.&proc	yes	C+[4] ; S–[3]
Bowen	meta-inf.	amalgam.	decl.&proc	yes	C– ; S–
3-LISP	meta-inf.	mono-ling.	proc.	yes	C– ; S–[3]
Socrates	task-man.	biling.	decl.&proc	no	C– ; S–
NuPRL	task-man.	biling.	proc.	no	C– ; S+
Proof Plans	task-man.	biling.	decl.&proc	no	C– ; S-

Legend:

?	=	unknown, cannot be determined from available literature.
_[n]	=	not applicable, see note n.
C+/–	=	completeness enforced/not enforced.
S+/–	=	soundness enforced/not enforced.

Notes:

[1] It is unclear how the object-oriented paradigm relates to the distinction declarative vs. procedural.

[2] The possibility of partial specifications does not occur in this system since it is a program built for a particular task, containing a full specification of one appropriate control regime.

[3] The system can be made combinatorially unsound because the confusion between object-level and meta-level language (mono-lingual) allows the meta-level predicates to introduce arbitrary bindings for the object-level variables.

[4] The system is only complete without the facility of dynamic cut-introduction.

Figure 3.5: properties of meta-level systems

3.3 Comparison of the different architectures

In this section we will compare the different types of meta-level architectures as distinguished in the previous section. We will identify shortcomings of almost all these types of meta-level architectures. This will lead us to prefer one particular type over all the others. However, even that type of architecture is not free from problems, but these will be discussed in more detail in chapter 6, and not in this section.

In our comparison of architecture, we will use as our main criterion the degree to which control knowledge can be separately and explicitly represented. This criterion is of course a direct consequence of the arguments in sections 1.2.1 and 1.2.2.

The most obvious problem is associated with the object-level-inference systems. The meta-level does not have a separate place in the architecture of these systems, and is not stated as explicitly as would be necessary in order to achieve the advantages discussed in the introduction (better explanation, re-usability and ease of development and debugging). The main structure of the control strategy of these systems is only implicit in the system. Although possibly available for inspection, it is never available for modification, and only a restricted number of aspects of the control strategy can be changed. For instance in the Gallaire/Lasserre system, it is possible to change the clause- and literal-selection strategies, but the fact that the system is chronologically backtracking and backward chaining is hardwired.

The mixed level systems do not suffer from this problem, but they have other problems associated with them, which can best be discussed using the subcategories of this type of system.

A problem that is associated with both the crisis-management systems and the reflect-and-act systems is that the search in the solution space is still performed at the object-level. As a result, the meta-level knowledge is only used as a preference criterion over the separate object-level search space, whereas in systems that are more meta-level driven the meta-level knowledge is used to specify completely the whole structure of the search space of the system. This means that no full advantage is taken from the fact that the meta-level search space is better behaved than the object-level search space.

A problem associated with the task-management systems is what could be called the black box effect: after the meta-level has decided on a task to be performed by the object-level, the object-level is no longer under the control of the meta-level, and again no full benefit is gained from the differences between meta-level and object-level search.

None of the mixed-level inference systems makes all the control knowledge in the system explicit: reflect-and-act systems deal only with conflict resolution strategies, crisis-management systems know how to solve impasses in the computation and subtask-management systems represent the selection of goals and subgoals, but none of them contains a full description of the object-level computation.

Meta-level inference systems do not suffer from these problems. In meta-level inference systems, the meta-knowledge is not just used as preference criteria over the separate object-level search space, but it is used to specify completely the whole structure of the search space. Meta-level inference systems perform their search in the meta-level search space and thereby gain the full benefit of the better properties of the meta-level search space. Furthermore, meta-level inference systems contain a full specification of the infer-

ence strategy of the system, thereby allowing the user to change any part of this strategy, and not just a few predefined aspects of it. However, meta-level inference systems do have problems of their own, which will be discussed in chapter 6.

This leaves us with the choice between the different subtypes of meta-level inference systems: mono-lingual, bilingual and amalgamated systems. A number of reasons can be given why it is important for the meta-level language to be separate from the object-level language, thus ruling out both the mono-lingual and the amalgamated systems.

- **Suitability:** First of all, there is an epistemological reason. [Reichgelt and van Harmelen, 1985,Reichgelt and van Harmelen, 1986] and [Chandrasekaran, 1983,Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987] argue that different domains require different representation languages. Since the object-level and the meta-level deal with widely different domains (the object-level deals with the application domain of the system, while the meta-level deals with the issue of controlling the object-level), it follows that these two levels of the system do indeed need different representation languages to suit their different needs.

- **Distinguishability:** A second argument concerns the modularity of the system. One of the advantages of separating control knowledge from domain knowledge is that the two can be changed independently. The same domain knowledge can be used for different tasks under different control regimes, and the same control knowledge can be used to solve similar tasks in different domains. However, this ability to vary the two levels independently would be greatly reduced if control knowledge and domain knowledge were represented together in one and the same language, which will inevitably lead to the two being represented in a mixed way, rather than separately, as needed to achieve the desired modularity.

- **Explanation:** The third argument is one about explanation. As argued in [Warner Hasling et al., 1984], the explanations given by reasoning systems should not only include *what* the system is doing, but also *why* it is doing a particular action and not another one. In other words: the control knowledge should be an identifiable part of the explanations given by the system. In order to enable the system to include control knowledge explicitly in its explanations, it is important for both the human reader and the automated explanation generator that control knowledge can be syntactically distinguished from domain knowledge. This would not be possible if one and the same language were used for both levels.

- **Formal correctness:** A final argument in favour of bilingual systems is the work in [Hill and Lloyd, 1988]. They provide a theoretical foundation for meta-programming in logic programming. After an initial attempt to use a mono-lingual system as the basis of their theoretical account (introducing a many-sorted language to rescue the separation of object-level and meta-level statements), they ultimately turn to a bilingual representation[4] as the only way of providing a satisfactory theoretical account of meta-level programming. This point is elaborated further in [Lloyd, 1988], as illustrated by the following quote:

[4]Hill and Lloyd use the term *"ground representation"* where we use bilingual representation, for the obvious reason that object-level expressions are represented by variable free terms at the meta-level.

> "[...], we argue that the current research effort being put into the [monolingual] interpreter and its enhancements is largely misplaced. It would be better to start with an interpreter based on the ground representation."

At this point we have to stress that a bilingual architecture does not imply that we can not use similar languages at object-level and meta-level if so desired. For instance, we could choose to use first order predicate calculus at both levels. The only requirement is that the two languages, although of the same type, are syntactically distinct. In other words, we would have two copies of the same language, one for each level, with the two copies being syntactically separate.

A further problem associated with the amalgamated approach is the recursive application of the naming convention. Each ground term in the meta-level language that is the name of an object-level formula is itself again an object-level formula (since the two languages are the same), and thus has some ground term as its name, etc. This introduces the possibility of self-referential sentences, which is necessary for introspection, or for incompleteness proofs à la Gödel (the self-referential capability is used in exactly this way in [Bowen and Kowalski, 1982]). However, if we are not interested in these aspects of meta-level reasoning this added complexity is not needed, and we can get away with the much simpler construction of separate languages.

Thus having narrowed our choice down to bilingual meta-level inference systems, we still have to discuss the best position on the other properties of meta-level systems described in the previous section: the possibility of partial specifications, a declarative versus a procedural meta-level language, and enforcing combinatorial soundness and completeness on the meta-level.

The possibility of writing only partial specifications of the control strategy of the system is obviously attractive for the development of a system. We can gradually refine the control strategy of the system, without overcommitting ourselves at any point, postponing decisions until we understand enough of the domain. However, a high price needs to be paid for this possibility, resulting in a severe restriction of the system's architecture. In order for the system to be able to "fill in the gaps" of the partial specification of the control regime by the user, it is necessary that this partial specification is of a particular format, so that it is possible for the system to identify which parts of the control regime are underspecified, and need to be filled in with default values. This restricts the possible range of control regimes that can be formulated by the user.

The question of a declarative versus a procedural meta-level language is not very clear. This issue will return again in chapter 5 where we will describe the meta-level architecture of the Socrates system.

Concerning the issues of combinatorial soundness and completeness we can say the following: for reasons discussed before, we would not want to enforce completeness on a meta-level architecture, since often the whole point of having a meta-level is to be able to avoid the expensive computation of certain object-level results. Combinatorial soundness on the other hand, not allowing the meta-level to extend the set of results that can be computed by the meta-level, is a desirable property, since we are only interested in the use of meta-level inference for control.

3.4 Conclusion

In this chapter we have categorised the meta-level systems described in the literature, and have distinguished the following types:

- object-level inference systems
- mixed-level inference systems, which can be divided into
  - reflect-and-act systems
  - crisis-management systems
  - subtask-management systems
- pure meta-level inference systems, which can be divided into
  - mono-lingual systems
  - bilingual systems
  - amalgamated systems

  Furthermore, a number of secondary properties of meta-level architectures were identified:
- Is the meta-level language declarative or procedural?
- Does the system allow partial specifications of the control regime?
- Does the system enforce combinatorial soundness and completeness of the control regime?

We have compared these systems, and have argued in favour of bilingual, pure meta-level inference systems.

Chapter 4

The structure of meta-level inference systems

In this chapter we will take a closer look at the structure of meta-level inference systems, the type of meta-level architecture that was argued for in the previous chapter. This will lead us to a list of necessary components for meta-level inference systems and for logic based meta-level inference systems in particular.

4.1 Basic components

At various places in the literature researchers have analysed the basic components of meta-level architectures in general (not just of meta-level inference systems). The most recent of these, which is based on previous analyses in [Smith, 1985] and [Batali, 1983], is given in [Maes, 1986b, Maes, 1986a]. Maes has suggested that introspective systems should have three essential ingredients, but in fact these requirements hold for any system with a meta-level component:

- a model of the object-level computation,
- a causal connection between meta-level actions and object-level behaviour,
- an architecture of introspection, which allows the system to switch between meta-level and object-level activities.

The most interesting of these (and as we will see the only one of interest for meta-level inference systems) is the model of the object-level computation. Before we discuss this in some detail, we will describe the other two points.

The requirement for a *causal connection* states that the model of the object-level computation should not only be inspectable by the meta-level, but the meta-level should be able to make changes in this model, and these changes should be reflected in the behaviour of the object-level. Smith [Smith, 1985] has introduced the words *introspective force* and *introspective faithfulness* for the two directions of the causal connection: "force" because changes made by the meta-level in the model of the object-level computation must be enforced upon the actual object-level machinery, and "faithfulness" because any change

of the object-level must be faithfully represented in the meta-level model of the object-level computation. A language such as OMEGA [Attardi and Simi, 1984] incorporates a 'read-only' model of the object-level program, having only introspective faithfulness, and lacking introspective force. Such a language would not have a proper bidirectional causal connection.

Maes describes the *introspection architecture* as

> "[an architecture which gives] the possibility to halt the computation of a program, jump to a reflective level where a model of that computation can be accessed and manipulated, and return to the affected computation afterwards."

The behaviour of such an introspection architecture corresponds to the behaviour of the mixed-level inference systems described in the previous chapter, where some criterion was used to switch activity between object- and meta-level.

The above description of essential components for meta-level systems is taken directly from [Maes, 1986b, Maes, 1986a]. Below we will specialise her analysis to meta-level inference systems in particular (which will result in removing two out of the three components), and then further specialise the remaining component for logic-based meta-level inference systems. This specialisation will allow us to give a much more detailed analysis of the required components of such a system.

Although the two components mentioned above (the causal connection and the introspection architecture) are part of meta-level systems in general, they are not required for meta-level inference systems. This is most clear for the introspection architecture. Since meta-level inference systems make all their inferences at the meta-level, and simulate the object-level computation at the meta-level using a full specification of it, there is no need to switch between the different levels: all the computation takes place at the meta-level. This makes the concept of the introspection architecture unnecessary for meta-level inference systems.

A similar argument holds for the causal connection. Its purpose is to ensure a correct relation between the object-level computation and the model of this computation at the meta-level. However, a meta-level inference system completely models the object-level computation at the meta-level, without separately executing any object-level computation. Since the model of the object-level computation is all there is, there is no need to worry about the force or faithfulness of this model, and thus the causal connection is also meaningless in meta-level inference systems.

This leaves us with the *model of the object-level computation* as the final component of meta-level architectures. This model is obviously of central importance in a system based on meta-level inference. After all, it is the subject of all the reasoning of the meta-level interpreter. Maes follows [Batali, 1983] in saying that the power of the meta-level is "model relative"[1], i.e. the power of the meta-level inference process is directly related to the richness of the model it has of the object-level computation. So far (in the previous chapter and above) we have said that meta-level inference systems have a full, complete model of the object-level computation, but this needs to be qualified. It is possible for

[1]Actually, Maes and Batali use the term "theory relative", but we prefer the term "model relative" instead, since the term "theory" already refers to object-level and meta-level theory.

meta-level inference systems to only have partial models of the object-level computation. In such a case some aspects of the object-level computation are not explicitly represented at the meta-level, but are instead implicit in the architecture of the system, and can therefore not be inspected or modified. This makes the power of a meta-level inference system relative to the completeness of its model of the object-level computation. In chapter 5 we will discuss a meta-level inference system whose meta-level model of the object-level computation is indeed not complete, and we will discuss the advantages and disadvantages of such an incomplete model.

Because this model of the object-level computation is of such central importance to meta-level inference systems, we will discuss it in some more detail. We can distinguish three components that constitute a model of computation:

- the object-level program,
- the computational behaviour of the object-level program,
- the state of the computation of the object-level program.

The first two of these components are static: the code is the set of instructions that form the object-level program, and the computational behaviour is the procedural interpretation under which these instructions will be executed. The third component of the model is dynamic: it represents the current state of the object-level computation, and changes as the computation proceeds. Each of these three components of the model of the object-level computation can be expressed at arbitrary levels of description. In principle it would be possible to give a description of voltages and electronic components of a particular machine, or, at the other extreme, in terms of a high-level language such as Prolog or LISP. Which level of description is appropriate depends on the particular application, but for applications in controlling the inference in reasoning systems we will be more interested in high level rather than low level descriptions.

It is important to note that this notion of a model of computation, consisting of these three components, is not in any way restricted to meta-level architectures. All computational devices can be modelled in this way at any level of description. For example, a computation in Pascal can be characterised by the program code, plus a description of the Pascal virtual machine (the Pascal interpreter), plus the state of this machine (as defined by the contents of the machine-registers and the procedure-call stack). The difference between meta-level architectures and arbitrary computational systems is that we require of meta-level architectures that all three components of this model (the program code, the computational behaviour and the state of the computation) are explicitly represented (i.e. available for inspection and modification), whereas in arbitrary computational systems (such as for instance a Pascal computation), we expect only the program code to be available for inspection and modification, while the other two components are implicit in the system.

4.2 Components of logic-based systems

As stated in chapter 1 we are particularly interested in systems that use logic as their representation language. The three components described above constitute a model of an

arbitrary computation that can be analysed in more detail in this specific context.

The most straightforward component is maybe the *program code.* In logic based systems this corresponds to the object-level theory (i.e. the set of object-level axioms), in knowledge-based systems also called the "knowledge base". The *state of the computation* in a logic-based system is completely specified by a proof tree, representing all inference steps that have been made so far.

The most complicated part of the model is the *computational behaviour.* In a logic-based system this consists of two elements: the inference rules, and information on how to use these inference rules. The set of inference rules for the object-level language defines the set of all possible computations that can be made (given a fixed object-level theory), but in order to specify what actual computations will be made, additional information is needed on how to use these inference rules. In other words, the inference rules (in conjunction with a given object-level theory) determine the search space of possible theorems, whereas a search strategy is needed to specify how this search space should be traversed. This distinction coincides with the one made in [Meltzer, 1971] where a theorem proving program is separated into an *inference system* and a *search strategy.* Meltzer writes $P = (I, \Sigma)$. The inference system I is defined by the set of inference rules, whereas the search strategy Σ consists of the strategies used for exploring the search space defined by the inference system. The explicit representation of the search strategy is at the very heart of the meta-level inference methodology. It constitutes the proof strategies that are axiomatised at the meta-level and they form the meta-level knowledge that the system possesses.

4.3 Completeness and soundness

It is important to emphasise that the inference system I determines only the theoretical space that should be searched by the system in order to find a solution. In general this space is infinite which prohibits its explicit representation. The search strategy Σ is needed to generate and traverse only relevant portions of this search space. The decomposition $P = (I, \Sigma)$ is used by Meltzer to distinguish two different notions of completeness, logical completeness and combinatorial completeness, where logical completeness is determined by I and combinatorial completeness by Σ. A theorem proving program P is *logically complete* if I is a complete set of inference rules for the logical language L of P (i.e. every valid sentence of L is provable under I). P is *"combinatorially complete"*[2] if every derivation admissible under I is eventually generated by P. In other words: if all theoretically possible proofs will also in practice be generated. Thus, logical completeness is a property of the object-level theory and the set of inference rules I, whereas combinatorial completeness is a property of the search strategy Σ as specified in the meta-level theory. Although Meltzer does not mention this, a weaker version of combinatorial completeness is also possible. A theorem proving program P is *weakly combinatorially complete* if every sentence provable under I will eventually be proved by P. This is a weaker requirement than the first (strong) definition of combinatorial completeness, since we only require that every provable sentence will eventually be proved, rather than requiring that every proof will eventually

[2]The notion of combinatorial completeness mentioned in chapter 3 is the same as the notion defined here.

be generated. Thus, if some sentences have multiple proofs, a weakly combinatorially complete system is only required to generate at least one of these proofs in practice, whereas a strongly combinatorially complete system must generate all of them.

In the meta-level inference system to be described in the next chapter we will see that it is possible to affect both the logical and the combinatorial completeness of that system (as determined by I and Σ respectively) at clearly separated places in the architecture.

The dual notions of logical and combinatorial completeness are of course logical and combinatorial soundness. The notion of combinatorial soundness has already been defined in chapter 3: a search strategy Σ is *combinatorially sound* if it only generates derivations admissible under the logical rules of inference I. The notion of logical soundness is as usual: I is *logically sound* if only valid sentences are provable under I. The definitions of logical soundness and completeness are of course relative to some intended semantics (defining the set of valid sentences). Similarly, the definitions of combinatorial soundness and completeness are relative to the logical derivability relation. Thus, even if a system is logically incomplete (i.e. not all valid formulae are logically derivable), it can still be combinatorially complete (if all logically derivable formulae are actually derivable in the system), and vice versa. The situation can be summarised as follows: if Π is the set of provable sentences (under I), T is the set of valid sentences (under some intended semantics), and Γ is the set of sentences whose proofs will actually be generated by Σ, then we can rephrase the definitions above as follows:

$$\begin{aligned} \text{logical soundness} &\leftrightarrow \Pi \subseteq \mathrm{T} \\ \text{logical completeness} &\leftrightarrow \mathrm{T} \subseteq \Pi \\ \text{combinatorial soundness} &\leftrightarrow \Gamma \subseteq \Pi \\ \text{combinatorial completeness} &\leftrightarrow \Pi \subseteq \Gamma \end{aligned}$$

4.4 Subcomponents of the search strategy

It is possible to further divide the search strategy Σ into three different components. A strategy consists of a *generative component*, a *directional component* and a *termination component*. inxxd$\Sigma_g$inxxd$\Sigma_d$inxxdΣ_t Following Meltzer, we could write $P = (I, (\Sigma_g, \Sigma_d, \Sigma_t))$. The generative component Σ_g of a strategy describes which part of the theoretical search space should be actually generated, the directional component Σ_d describes how the resulting space should be traversed during the search process, and the termination component Σ_t decides when the search process should be stopped. In the context of logic-based inference engines, Σ_g tells us how to expand a given node in the AND/OR tree, while Σ_d tells us which node should be chosen for the continuation of the search process. Σ_g governs decisions such as which subset of the set of inference rules is to be used at a particular node to generate new nodes, whether these inference rules are to be used one at a time or all at once, whether they should be used exhaustively or not, etc. Σ_d deals with decisions about the selection and ordering of nodes in the search tree. Σ_t determines how many nodes will be visited, how many solutions will be generated, if there is a maximum limit on the amount of effort spent by the system, etc. As said above, the combinatorial completeness of a system is determined by its search strategy Σ. Actually,

feature of search process	component of heuristic
exhaustive/ non-exhaustive	termination & generative
a maximum search depth	termination & directional
branch and bound	generative
best first/ breadth first/ depth first/ agenda based	directional
forward/ backward	generative
iterative-deepening	termination & directional

Figure 4.1: examples of components of strategies

for systems with a finite search space, only Σ_g and Σ_t affect the combinatorial completeness, and Σ_d only affects the order in which solutions are generated. However, for systems with infinite branches in the search space, Σ_d will also affect the completeness, since Σ_d will determine whether or not the system manages to avoid getting trapped in the infinite branches of the search space (this property is sometimes called fairness). (See [Hogger, 1984] for a precise definition of fairness).

Under these definitions the set of inference rules I, plus the choice between forward or backward use of them, determines which solutions a system can generate in principle; Σ_g, Σ_t (and possibly Σ_d) determine which of these theoretically possible solutions will be actually generated in practice; and Σ_d determines in which order these solutions will be generated. Figure 4.1 lists a number of different properties of control regimes, and shows how these properties are defined by either the generative, the directional or the termination components of strategies.

To illustrate the distinction between Σ_g, Σ_d and Σ_t, we can compare three related Prolog systems, namely a normal Prolog interpreter, the enhanced Prolog system from Gallaire and Lasserre (as described in section 2.4.1) and the bilingual meta-circular Prolog interpreter that will be used in chapter 6, figure 6.4b, which is repeated here in figure 4.2 for convenience. A standard Prolog interpreter obviously fixes all three of $\Sigma_{g,d,t}$: Σ_g is fixed to generate the full search space (i.e. no control decisions are taken by disregarding certain parts of the search space); Σ_d is fixed to Prolog's standard top-to-bottom, left-to-right execution rule (textual order); Σ_t is fixed to terminate the search process when one solution has been found. The enhanced Prolog system by Gallaire and Lasserre fixes Σ_g and Σ_t to the same choices, but allows the user to define the choices for Σ_d (within certain limits set by the language made available for this purpose): the user can affect

```
solve([], []).
solve([G|Gs], S) :-
    get_clause(G, C),
    rename_vars(C, [G|Gs], C1),
    head(C1, H1),
    unify(G, H1, S1),
    body(C1, B1),
    append(B1, Gs, NewGs),
    instantiate(NewGs, S1, NewGsPlusS1),
    solve(NewGsPlusS1, S2),
    compose(S1, S2, S).
```

Figure 4.2: code of a bilingual meta-circular Prolog interpreter

the choice-strategy for clauses and goals (i.e. at OR-GOALS and AND-GOALS). Finally, the bilingual meta-interpreter from figure 4.2 allows the redefinition of all three components. Σ_g is embodied in the definition of `get_clause/2`. This definition could be adapted to not generate all clauses matching G, but to leave some out depending on some criterion programmed by the user. Σ_d is distributed over two places in the code: the directional strategy for OR-NODES is again embodied in `get_clause/2`, and could be changed there. The directional strategy for AND-NODES is represented by the facts that (1) the 2nd clause of `solve/2` always selects the first conjunct of the list of unproven conjuncts, and (2) new conjuncts are always prepended to the list of remaining conjuncts, using `append/3`. If the beginning of the 2nd clause for `solve/2` were changed to

```
solve(Gs, S) :-
        get_conjunct(G, Gs),
        ...
```

then all of the Σ_d for AND-NODES would be represented by `get_conjunct/2`. Finally, the termination component Σ_t of the bilingual meta-interpreter is represented by the first clause of `solve/2`, indicating to stop as soon as we have no more subgoals to prove. It should be noted that all three of these systems do not allow their inference component I to be changed. The next chapter will describe a system which not only allows variation of all three of $\Sigma_{g,d,t}$, but also of I.

4.5 Conclusions

Summarizing, we can represent the above analysis of the components of a logic-based system based on meta-level inference as in figure 4.3. From this we see that the essential components are a tuple $(O, I, \Sigma_g, \Sigma_d, \Sigma_t, T)$[3]:

[3]Note that with these definitions, the model of the computation corresponds closely to what Hayes calls a "state" (see section 2.3.1).

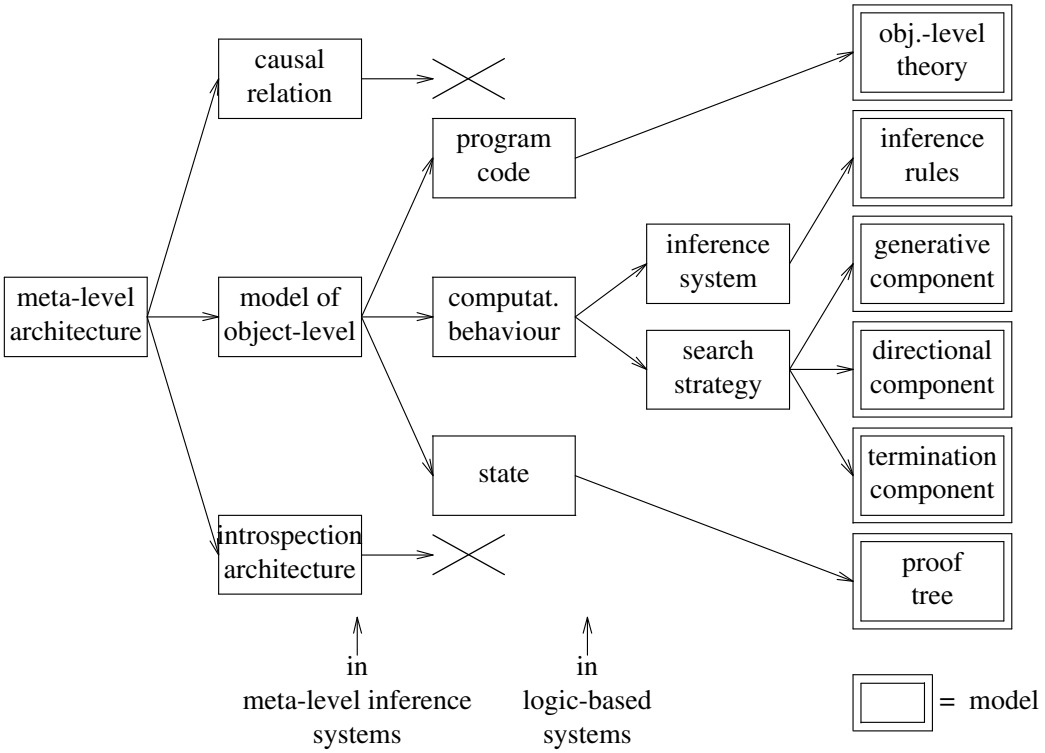


Figure 4.3: components of logic-based meta-level inference systems

- an object-level theory O
- a set of inference-rules I
- information on how to use these inference-rules, Σ, consisting of
  - a generative component Σ_g
  - a directional component Σ_d
  - a termination component Σ_t
- a representation of the object-level proof tree T.

The next chapter will describe the architecture of a meta-level inference system in which all these components are explicitly represented and clearly recognisable.

Chapter 5

A case study of a meta-level inference system

In chapter 3 we argued in favour of one type of meta-level architecture, the so-called bilingual meta-level inference systems, and we discussed the structure of these systems in the previous chapter. The goal of this chapter is to look at one such system, called Socrates, in some detail, to illustrate the decisions involved in building it, and how these decisions influence the behaviour of the system. The first section of this chapter will describe the Socrates meta-level architecture, the second section will locate the Socrates architecture in the classification given in the previous chapter, and a final section will discuss some of the choices that were made while building Socrates, and will anticipate some of the problems and solutions discussed in later chapters.

5.1 The Socrates architecture

The Socrates system, described in [Reichgelt and van Harmelen, 1987] and [Corlett et al., 1988], is an attempt to create an environment for building knowledge-based systems based on the following principles:

- An epistemological analysis of the domain and task of a particular application guides the choice of the appropriate representation language and the appropriate control regime. As argued in [Reichgelt and van Harmelen, 1985,Reichgelt and van Harmelen, 1986] this means that different application domains will need different representation languages and that different tasks will need different control regimes. Similar arguments are made in [Chandrasekaran, 1983, Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987] and [Breuker and Wielinga, 1986].

- Logic is used as the representation formalism. The case for logic as a representation language was argued in chapter 1. This choice for logic is not in conflict with the first point (which claimed that different domains would need different representation

[0]The work on the Socrates meta-level architecture reported in this chapter was done jointly with Han Reichgelt and Peter Jackson in the Alvey funded project "A Flexible Toolkit for Building Expert Systems".

languages), since the language of logic is not restricted to that of standard two-valued, truth functional, first order predicate calculus. Many other logics have been proposed, offering a wide range of expressional and inferential power.

- Control knowledge is represented explicitly and is separated from the domain knowledge. The case for this was also argued in chapter 1, and will not be repeated here.

When configuring the Socrates environment for a particular application, its architecture can be varied along three dimensions:

- the representation language: users can define their own logical representation language, including normal first order logic (possibly many-sorted), modal logics, temporal logics, etc.
- the inference rules for the logical language, the set of rules that determine the possible inferences made in the logical language, can be changed by the user.
- the control regime under which the inference rules will be used to perform proofs in the logical representation language.

It is precisely the last two points which provide the meta-level architecture of Socrates. We will discuss these two points in more detail. For a detailed description of the first dimension of flexibility, the definition of a logical representation language, the reader is referred to [Corlett et al., 1988]. However, in order to allow a full explanation of the meta-level architecture we briefly describe the first dimension of flexibility (the representation language) here.

Socrates allows the user to define the logical language that is to be used for representing the domain knowledge. For this purpose the user declares the set of predicate symbols to be used in the logical language, as well as the sets of function symbols and constants of the language. Furthermore, the user defines the logical connectives of the language, (conjunction, implication, modal operators, etc). On the basis of these declarations, the system configures a storage and retrieval mechanism for expressions in this logical language. The user can enter formulae of the logical language in a "knowledge base" (theory), or remove them again. Furthermore, a retrieval function is constructed which takes as its input a formula and a theory-name, tries to unify the input formula with a formula in the theory, and returns the resulting variable bindings. Other features of the Socrates representation mechanism such as

- a programmable indexing mechanism for formulae
- a hierarchical partioning mechanism for theories
- a type-lattice for many sorted logics
- the use of more than one logical language
- a slot-value annotation mechanism for formulae

are not essential for understanding the Socrates meta-level architecture, and are therefore not discussed here. The next two subsections will discuss the two aspects of the Socrates meta-level architecture mentioned above, namely the declaration of a proof theory (the rules of inference) and of a proof strategy (the control regime).

5.1.1 Declaration of a proof theory

Given the syntax of a logical language that will be used as a representation language, the user needs to specify how expressions in this logical language can be manipulated to derive new formulae from old formulae, in other words: how we can perform proofs with expressions of the logical language. A set of logical inference rules is defined for this purpose. Since the system is based on the natural deduction style of theorem proving [Prawitz, 1965], inference rules take the form of, for example

$$P, P \rightarrow Q \vdash Q$$
$$P, Q \vdash P \wedge Q$$

As shown in this example, inference rules are expressed in a meta-logical language very close to that in standard logical use. P and Q are propositional variables (i.e. ranging over logical propositions). These inference rules can be used in both a forward and a backward direction. For instance, Modus Ponens (the first of the two rules above) can be used to determine that it suffices to prove P and $P \rightarrow Q$ in order to prove Q (backward use), or the rule can be used to infer that Q is true when we know that both P and $P \rightarrow Q$ are true (forward use). Which of these two directions should be used is a control decision, and is therefore a meta-level issue, which is not decided as part of the proof theory, but as part of the control strategy defined at the meta-level.

There are a number of reasons why Socrates follows Bledsoe [Bledsoe, 1981] in using the natural deduction style of performing proofs rather than, for instance, resolution. Although natural deduction systems use a relatively large number of inference rules (as opposed to the single inference rule of resolution based systems), and thereby create a potential control problem, the following arguments can be given in favour of natural deduction. The inference rules of a natural deduction system are more intuitively meaningful than for instance the resolution rule, and no normal forms are required for the formulae used in a proof. As a result, the proofs performed by a natural deduction system are easier to follow for a human reader, thereby improving the possibilities for explanation facilities based on these proofs. The naturalness of the proof development also makes it easier to write heuristics to control the problem solving process in a meta-level control strategy.

Not all the inference rules that the system uses are declared as part of the proof strategy. Some inference rules are incorporated into the retrieval mechanism instead. Incorporating an inference rule in the retrieval mechanism means that the retrieval function described above will not only try to unify its input formula with formulae in the theory, but that it will also try to prove the input formula from the theory using the incorporated inference rule. A first set of inference rules that is incorporated into the retrieval mechanism are the rules that tell the system how to deal with quantification. These rules:

$$\begin{aligned} \forall x P[x] &\vdash P[c] \\ P[c] &\vdash \exists x P[x] \\ \exists x \forall y P[x, y] &\vdash \forall y \exists x P[x, y] \end{aligned}$$

are taken to be of universal validity (that is: across different application areas of the system), and are therefore hardwired into the retrieval mechanism. A second set of rules

can be made part of the retrieval mechanism rather than the proof theory, namely the rules that deal with the commutativity and associativity of certain logical connectives, e.g.:

$$\begin{aligned} P \wedge Q &\vdash Q \wedge P \\ (P \wedge Q) \wedge R &\vdash P \wedge (Q \wedge R) \end{aligned}$$

The main reason for taking these rules out of the explicit declaration of the proof theory is efficiency. When users declare a logical representation language, they can declare logical connectives to be commutative and/or associative, if so desired. This will then result in either or both of the above rules of inference being incorporated in the retrieval mechanism. Because these rules can be incorporated in the retrieval mechanism, they can be taken into account by the unification algorithm of Socrates. Thus, when proving a query that can be derived from the theory using one of the hardwired inference rules, no inference steps need to be taken by the inference machinery (object-level and meta-level), but the unification algorithm that is used for retrieving items from the theory makes these limited deductions. As an example of the effectiveness of these hardwired inference rules, let us assume that the connective $\wedge$ has been declared commutative (as in the first rule above), and that we are trying to prove the object-level goal $f(a,b) \wedge g(a,b)$, when the object-level axiom $\forall x[g(a,x) \wedge f(a,x)]$ is in the object-level theory. The retrieval mechanism can deduce the goal directly from the axiom by using two of the inference rules mentioned above (namely universal instantiation and commutativity of $\wedge$). Without these two rules, the interpreter would have to make two cycles to prove the same goal. This saving easily offsets the increase in cost of the more complex retrieval mechanism.

On the basis of the user's declarations of the logical language, the properties of the connectives, and the rules of inference, Socrates automatically constructs the following two predicates that can be used by the user in the definition of a proof strategy (which will be described in the next section):

- `object_level_axiom(F, T, S)`: this predicate will be true if there is a substitution `S` for variables occurring in the object-level formula `F` such that applying `S` to `F` results in an axiom in the object-level theory `T`.

- `object_level_inference(F, T, D, L)`: this predicate will be true if `L` is the list of formulae which can be constructed by applying an inference rule of the object-level theory `T`, in direction `D` (i.e. forward or backward) to the object-level formula `F`. For example, if in a theory `t1`:

  `t1 = {f(1)→ f(10), f(2)}`

  the rules Modus Ponens and Conjunction Introduction have been declared, then the call:

  ```
  object_level_inference(f(1), t1, forward, L)
  ```

  would succeed with

  `L = [f(10), f(1)∧f(2),f(1)∧f(1)→f(10)]`

A final point to be made about the declaration of the proof theory concerns the logical soundness and completeness of the set of inference rules. In order to guarantee soundness[1]of the proof theory, the user should not be allowed to declare arbitrary inference rules, but only to select inference rules from a predefined (and sound) set. (Such a selection procedure has not been provided in the current implementation of Socrates). This selection process will of course affect the logical completeness of the system. However, the loss of completeness in the context of knowledge-base systems is not serious, since one does not want to infer *all* facts that are logically implied by the available knowledge, but only those facts that one is interested in.

5.1.2 Declaration of a proof strategy

Given a logical representation language, and having defined the set of inference rules for this language, the system is completely specified from a logician's point of view. The declaration of a logical language and its proof theory completely determines all possible inferences that the system can make. However, in order to create a practical computer system, a specific control strategy has to be defined, which will specify how the space of possible proofs will be explored by the system. This separation represents exactly the analysis of the previous chapter of a system P into a logical inference system I and a search procedure Σ. The declaration of the proof theory corresponds to defining I, while the declaration of a proof strategy corresponds to defining Σ.

The language that is used to express the control strategy is Horn Clause Logic [Horn, 1951]. This language, although also a logical representation language, should be distinguished from the logical languages used to represent the domain knowledge. Unlike the object-level languages, the language used at the meta-level has a fixed set of logical connectives, namely exactly those connectives needed in Horn Clause Logic (conjunction, implication and negation) plus disjunction. All these connectives are declared as non-commutative, non-associative. This is done because the procedural interpretation (i.e. the way in which the meta-level interpreter executes expressions of the language) is also fixed. The procedural interpretation of the language is the standard interpretation for Horn Clauses, the standard depth-first proof procedure as found in Prolog systems. The reason why Socrates does not allow the user to change the control regime of the meta-level interpreter (which would amount to providing a meta-meta-level interpreter) is based on the previously mentioned idea that an epistemological analysis of domain and task of a particular application should guide the choice of the appropriate representation language and the appropriate control regime: typical tasks such as diagnosis, planning, monitoring etc. are related to particular control regimes (as argued in [Reichgelt and van Harmelen, 1985, Reichgelt and van Harmelen, 1986] and [Chandrasekaran, 1983, Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987]). The meta-level controls the behaviour of the object-level interpreter according to the task to be performed. The variation in control is achieved by changing the meta-level theory. There is no need to change the interpreter which always has the same task, namely controlling the behaviour of the object-level interpreter by using the data in the meta-level theory[2].

[1]Where soundness is defined with respect to some intended semantics imposed by the user.

[2]This separation of task from domain, with only the task influencing the control strategy is of course

With the logical connectives and their procedural interpretation fixed, the only parts of the logical meta-level language that are subject to declarations of the user are the set of constants, predicates and function symbols, and the set of *evaluable predicates*. These predicates are somewhat similar to what Weyhrauch [Weyhrauch, 1981] calls *semantic attachment*. When one of these predicates is encountered in a proof the system will execute a procedure defined for this predicate to determine its truth value and possibly provide bindings for any variables. These predicates provide an interface between the logical language and the computational environment of the system, enabling external systems interaction, input/output for interacting with the user, and the access of the facilities provided by the implementation language of the system. Below we will see that this mechanism is also used to implement the communication between object-level and meta-level.

Thus, Socrates allows the user to specify a control strategy in a logical language. This control strategy provides the system with a description of its desired behaviour. This description is interpreted at run time by the meta-level interpreter. As a result, the meta-level interpreter executes this control strategy, and thereby guides the search through the space of all possible proofs.

Figure 5.1 shows an example of a description of a local-best-first, non-exhaustive, backward-chaining, control strategy. In this example, clause [1] states that in order to prove a non-compound expression F on the basis of the contents of theory named P giving a substitution S as a result, the system should either try to see if the formula is a known fact in the theory, or the system should try to infer the formula on its own, or it should ask the user. Trying to infer the formula means generating all possible inferences, selecting some of these possible inferences, and continuing with clause [2]. This clause chooses the best of all selected possible steps, and tries to continue the proof with this selection. If this succeeds, the proof terminates (i.e. is non-exhaustive), if this fails, the proof continues with the next best step. Clause [3] states the criterion used in the best-first search, while clause [4] describes what needs to be done in order to prove a compound expression (that is: a formula containing the meta-level function-symbol '`,`' as used in the definition of inference rules): prove both left- and right-hand sides of the compound expression, and combine the results.

This example shows how the different aspects of this strategy can be changed if needed for a particular application. For example, the order of the disjuncts in clause [1] might be changed to ask the end-user for solutions before the system tries a proof itself, or the `ask-user` disjunct might be deleted all together. The criterion used for best-first scheduling could be changed, or a new decision for scheduling the order in which conjuncts are proved in clause [4] could be introduced. More thorough changes to the strategy could also be made, but they would amount to writing a completely new proof strategy, rather than changing the one shown in this example.

The only predicates which are predefined by the system (and therefore fixed) are the predicates `compound-expression`, `object-level-axiom` and `object-level-inference`, and the last two of these are constructed on the basis of the object-level declarations by the user. As a result, all aspects of the example strategy shown in figure 5.1 can be redefined

an idealisation. In a closer analysis (such as in [Chandrasekaran, 1983, Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987] and in [Breuker and Wielinga, 1986]) there is an interaction between the two.

```
[1]    (∀ F, P, S)
       [¬ compound-expression(F) ∧
        (   object-level-axiom(F, P, S)
         ∨  infer(F, P, S)
         ∨  ask-user(F, S)
        )
        → proof(F, P, S)
       ]

[2]    (∀ F, P, Best, Next, Rest, S)
       [¬ compound-expression(F) ∧
        object-level-inference(F, Next, backward) ∧
        best(Next, Best, Rest) ∧
        (   proof(Best, P, S)
         ∨  infer(Rest, P, S)
        )
        → infer(F, P, S)
       ]

[3]    (∀ List, Rest, Best)
       [highest-certainty-value(List, Best, Rest)
        → best(List, Best, Rest)
       ]

[4]    (∀ F, P, Lhs, Rhs, S, LhsSubst, RhsSubst)
       [compound-expression(F) ∧
        split-compound-expression(F, Lhs, Rhs) ∧
        proof(Lhs, P, LhsSubst) ∧
        proof(Rhs, P, RhsSubst) ∧
        combine(LhsSubst, RhsSubst, S)
        → proof(F, P, S)
       ]
```

Figure 5.1: specification of control in Socrates

by the user.

In order to achieve the flexibility described above (the ability to vary both the logical inference rules and the control strategy), Socrates is implemented as a two-layered architecture. The bottom layer of this architecture is the object-level, and embodies the declarations made for the representation language and its inference rules (the proof theory). This object-level consists of a unifier for object-level expressions that takes into account the declarations concerning associativity, commutativity and quantification, plus a mechanism

for storing and retrieving expressions in the logical language that the user has specified. The predicate `object-level-axiom` in figure 5.1 represents this retrieval mechanism. Furthermore, and most importantly in the context of meta-level reasoning, the object-level contains a procedure that takes as its input an expression in the object-level logical language plus a direction in which to apply the inference rules (either `forward` or `backward`), and returns as output a set of formulae that can be derived from the input formula by applying all the declared inference rules. This is the predicate `object-level-inference` in the example of the control regime in figure 5.1. This predicate models the object-level proof theory at the meta-level, and provides the meta-level interpreter with an explicit representation of the object-level search space. This enables the meta-level interpreter to manipulate this search space, and to choose which branches of the object-level proof tree will be expanded, on the basis of the control regime provided by the user.

We can use this example of a control regime formulated in Socrates to illustrate the subdivision of a search strategy Σ into its three components Σ_g, Σ_d and Σ_t, as described in the previous chapter. In figure 5.1, axioms [1] and [2] embody the generative component of the heuristic Σ_g, describing how a node should be expanded, whereas axiom [3] is the directional component Σ_d, specifying which node to choose for expansion. Axiom [4] is a mixture of both, specifying how to expand a `compound-expression`-node, but also specifying which part of the expansion to pursue first, the left- or the right-hand side of the compound expression. This axiom could have been written as

```
[4a]   (∀ F, Lhs, Rhs, First, Second, S, LhsSubst, RhsSubst)
       [compound-expression(F) ∧
        split-compound-expression(F, Lhs, Rhs) ∧
        select-conjunct(Lhs, Rhs, First, Second) ∧
        proof(First, FirstSubst) ∧
        proof(Second, SecondSubst) ∧
        combine(FirstSubst, SecondSubst, S)
        → proof(F, S)
       ]

[4b]   (∀ Conj1, Conj2, First, Second)
       [...  some criteria for selecting a conjunct ...
        → select-conjunct(Conj1, Conj2, First, Second)
       ]
```

in which case axiom [4a] strictly deals with generation and [4b] with direction. Finally, the fact that disjunction is used in axioms [1] and [2] to string together the different ways of achieving a goal embodies the termination heuristic Σ_t: only one solution (non-exhaustive search) is required before the search for a proof terminates.

Although not crucial to the meta-level architecture of Socrates, it is worth mentioning that the system provides a third layer on top of the object-level and meta-level described above. This third level is called the *scheduler*, and it is intended to deal with the notion of a *subtask*. As shown in [Reichgelt and van Harmelen, 1986], many expert systems perform not just one simple task, but a composite one that can be thought of as consisting of a number of elementary tasks (MYCIN, R1 and VM are among the systems discussed in

that paper). It is unlikely that one appropriate control regime can be found that would be suitable for these composite tasks. Rather, the composite task should be split up into its constituent subtasks, and a proper control regime can then be chosen for each of the subtasks. The subtasks that would be the end results of this decomposition process are the kind of prototypical tasks proposed in [Reichgelt and van Harmelen, 1986], [Chandrasekaran, 1983, Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987] and [Breuker and Wielinga, 1986] like classification, monitoring, simulation, design etc. The scheduling level of the Socrates architecture is meant to deal with this subdivision of the major task into prototypical subtasks. Each of these prototypical subtasks can then be solved using the appropriate meta-level control strategy. (By using a partitioning mechanism, it is possible to equip a Socrates configuration with more than one control strategy). For engineering purposes it would be easiest to equip the scheduling level with a language similar to (but again syntactically separate from) the language used to describe the control strategy at the meta-level, because all the machinery to deal with logical languages (unification, substitution etc) is already available for the object- and the meta-level. However, early experience indicated that the type of knowledge to be expressed at the scheduling level is of a very procedural nature (even more so than the knowledge expressed at the meta-level), and therefore a language with more conventional procedural primitives such as sequences, conditionals, loops and subroutines was thought to be more appropriate. The relation between meta-level theory and scheduler is very different from the relation between object-level and meta-level theory. The scheduler does not specify the complete control strategy for the meta-level theory (as the meta-level theory does for the object-level theory). It only selects an appropriate meta-level theory to be executed, but the way in which this theory is executed cannot be influenced by the schedule: the procedural interpretation of the meta-level Horn Clause theory is fixed. In terms of the classification of chapter 3.1, the relation between scheduler and meta-level theory is like a subtask-management system (section 3.1.2.3).

5.1.3 Practical experience with Socrates

The Socrates system as described above has been used in a number of applications. The main goal of these applications was to validate the claim that the Socrates architecture would be flexible enough to cope with a number of different tasks in a number of different domains. The following is a list of projects for which the Socrates system has been used. A more detailed description of these can be found in [Jackson et al., 1989, chapter 4].

- OSIRIS is a system built with Socrates, designed to assist maintenance programmers in the task of debugging a particular, large, real-time software system. On the basis of error messages produced by the real time software, OSIRIS uses some 130 object-level axioms to predict which modules of the system might be at fault, using certainty weightings to deduce the likelihood of failure for each module. The OSIRIS system uses a backward-chaining strategy to do its diagnosis task.

[2]Work on the Socrates applications was done jointly with Robert Corlett, Nick Davies and Robin Khan from GEC-Marconi Research Centre.

- DOCS is a system built with Socrates, and is a re-implementation of an existing system that assesses the risk category of patients with chest pains arriving in a hospital. The DOCS system used some 130 object-level rules and some 80 sorts in the sorted object-level logic to perform diagnoses using two different control strategies at the meta-level, one for easy and one for hard cases. The logical reformulation of the existing system in Socrates plus a verification theorem prover formulated as a meta-level strategy were used to detect inconsistencies in the object-level theory, and several such inconsistencies were indeed discovered.

- A system has been built with Socrates to do a configuration task: it allocates a limited set of resources to a set of tasks under constraints given by the user. An eager forward chaining strategy was used for this task.

- A path-finding system was built to find shortest paths in a network, using an agenda based control mechanism, with sub-goals picked from the agenda for further processing on the basis of a small number of heuristics.

- A challenge problem for theorem provers known as Schubert's Steamroller [Walther, 1984] has been successfully and efficiently solved by Socrates using a many sorted formalisation of the problem, and a number of different control strategies: breadth-first search, best-first search and alpha-beta pruning.

- A modal temporal logic has been implemented in the Socrates framework [Reichgelt, 1987], although this language has not yet been used in a practical application.

- A default logic has been implemented in the Socrates framework [Davies, 1988], but again this has not yet been used in a practical application.

The above lists demonstrates that the flexibility of Socrates with regard to its object-level language and its meta-level control strategies make it applicable to a wide class of varying tasks and domains. This success has to be qualified by saying that the performance of these applications in terms of speed would not have been sufficient to make them suitable as usable systems in a production environment. Later chapters will discuss this efficiency problem extensively.

5.2 Implementation

This section will briefly discuss some technical aspects of the implementation of the Socrates system, with emphasis on the meta-level control component of the system.

Implementation language: After an initial implementation on the InterLisp-D machine (whose programming environment facilitated the prototyping process), a final version was implemented in CommonLisp on a Sun workstation. This system consisted of some 10,000 lines of CommonLisp code, divided roughly in 6000 lines for object-level representation and knowledge base management[3], and some 4000 lines for meta-level control.

User Interface: [4] An additional 2000 lines of CommonLisp code were written in the Poplog CommonLisp system to provide a window/menu-based environment for Socrates implemented using the Poplog CommonLisp window manager. Interaction with all components of the system (declaration of object-level language and inference rules, construction of object-level and meta-level theories, execution of the resulting system) was all done through a series of protocols and menus which guide the user through the correct sequence of definitions and declarations.

Meta-level theories: Apart from the specialised meta-level theories written for some of the applications listed above, a number of weak, general problem solving methods were encoded as a library of meta-level theories. This library included the following search strategies: depth-first, breadth-first, local-best-first, exhaustive and non-exhaustive versions of these three, branch and bound search, iterative deepening and interleaved forward and backward search. All these search strategies were written in the Horn Clause logic available as the meta-level language, and all were independent of the particular object-level language and rules of inference that they would have to control, thus validating the point about the reusability of control-knowledge made earlier.

Evaluable predicates: The mechanism of evaluable predicates was initially implemented only to enable the communication between meta-level and object-level (the automatically constructed predicates `object-level-axiom` and `object-level-inference` are implemented as evaluable predicates), and to allow inspection of meta-level names of object-level formulae (see section 5.4.4 for a more detailed discussion of this topic). It was however quickly recognised that this facility of interfacing predicates in the meta-level theory to computation in the underlying implementation language was a useful tool in general, and it was made available for general use in writing meta-level theories. A small programming environment guaranteed the correct interaction between CommonLisp code and the logical meta-level language, and allowed the compilation of these evaluable predicates by the Lisp compiler. Specific use of this facility was made for implementing arithmetic in the meta-level language, and for user interaction in the meta-level theory (as illustrated by the predicate `ask-user` in figure 5.1). Potential other uses include the interaction with external devices and window management (which would be part of a meta-level theory that implemented more cooperative problem solving strategies).

Machine constructed predicates: An interesting part of the Socrates code was concerned with the automatic generation of the predicates `object-level-axiom` and `object-level-inference`. The behaviour of both these predicates is determined by a number of definitions made by the user. For the predicate `object-level-axiom`, these definitions are the contents of the object-level theory and the syntax of the object-level language, and for the predicate `object-level-inference` these definitions are the set of inference rules and again the syntax of the object-level language. It is quite acceptable for these predicates to inspect the definition of object-level theory and inference rules during

[4] These items were implemented by Robert Corlett, Nick Davies and Robin Khan from GEC Software Research. The other items were implemented by the author.

their execution, but it would be prohibitively inefficient if these predicates continuously had to access the definitions of the object-level language at run-time. This would have to be done every time a unification was attempted between two object-level formulae or between an object-level formula and a rule of inference. To avoid this problem, the Socrates system automatically constructed a unification procedure after the user has completed the definition of the object-level language. This construction process takes into account the associativity and commutativity properties of the object-level connectives, the declarations of the sort lattice for the object-level language, and the definitions of predicates, function symbols and constants made by the user. The result of this construction process is a CommonLisp procedure that performs efficient unification of object-level expressions which can, through the Lisp compiler, be translated down into machine code for optimal performance. This construction process is very close to partial evaluation (discussed in chapter 7): the general definitions of the meta-level predicates are specialised for the particular values of object-level theory and proof theory.

Meta-level interpreter: A number of different architectures were used for the Socrates meta-meta-level interpreter (i.e. the code which interpreted the Horn Clauses of the meta-level theory). Some of the issues involved are discussed below:

- The first implementation consisted of a stack-based interpreter, where the Horn Clause recursion stack was maintained as a Lisp data-structure. The advantage of this organisation was that all data-structures of the interpreter were inspectable by the user for explanation and debugging purposes. However, the performance of this implementation was unacceptably low, with its speed (in terms of LIPS[5] ratings) dropping to single figures.

- To cope with this problem a second version was implemented which did not represent the Horn Clause stack as a Lisp data-structure, but instead used the Lisp recursion stack for this purpose. As a result, the debugging options for the user of this interpreter were greatly reduced, but performance improved to LIPS ratings in double figures.

- A final version dropped the stack based architecture of the interpreter altogether, and instead used an interpreter based on the notion of continuation passing very similar to the FOOLOG system described in [Nilsson, 1983,Nilsson, 1984]. This further improved the performance to a level which was sufficient to realise the applications mentioned in the previous section. However, performance of the interpreter remained a bottleneck in Socrates, and this problem will be discussed in greater depth in the following chapters.

Current status: After the work on Socrates had been reported in the literature (e.g in [Corlett et al., 1988] and, most extensively, [Jackson et al., 1989]), the Socrates program did not continue to be used in practice. As mentioned above, the version with the best engineered user-interface was built in Poplog CommonLisp, and as the Poplog window

[5]The LIPS measure (Logical Inferences Per Second) will be discussed extensively in the next chapter.

manager was substantially redesigned, no manpower was available to upgrade Socrates to the new environment. As a result no working version of Socrates exists at the moment.

5.3 Socrates in the classification of meta-level architectures

The main distinction made in chapter 3 between different types of meta-level architectures was based on their locus of action. Of the three possible categories (object-level inference systems, mixed-level inference systems and meta-level inference systems), Socrates classifies as a meta-level inference system. In the architecture described above the main activity takes place at the meta-level, while the object-level proofs are simulated at the meta-level. Superficially it would seem that Socrates is a mixed-level inference system, since it has an explicit representation of the object-level theory rather than a meta-level encoding of the object-level theory (as PRESS does). However, the separate representation of the object-level theory can be regarded as a notational variant of a series of n meta-level axioms of the form

`object-level-axiom(`F'_1`)`
...
`object-level-axiom(`F'_n`)`

one for each object-level axiom F_i, with each F'_i the meta-level name of an object-level axiom F_i. Given such a meta-level encoding, the two predicates that access the separate object-level theory (`object-level-inference` and `object-level-axiom`) could have been explicitly defined at the meta-level, and the system is then clearly a meta-level inference system. Within the category of meta-level inference systems, Socrates is of the bilingual variety, rather than being mono-lingual or amalgamated, since Socrates uses different logical languages for meta- and object-level. Even when the object-level representation language happened to be defined as Horn Clause logic (the same language as used at the meta-level), the two languages would still be syntactically separate. The two languages relate to each other in a specific way. The meta-level language contains names for object-level expressions. In Socrates, the name of an object-level formula is a constant of the meta-level language. Other meta-level constants are used to denote bindings of object-level variables (the results of object-level proofs). Furthermore, meta-level expressions can range over other extra-logical properties of object-level expressions such as truth values, certainty factors, justifications etc. In this way Socrates could for instance be configured to deal with certainty values by specifying as part of the control strategy how certainty values should be used in a proof. This corresponds to the approach suggested in [Shapiro, 1983], with the important difference that Socrates makes a correct distinction between meta-level and object-level languages, whereas Shapiro mixes the two, and uses Prolog for both.

A feature worth emphasising is that Socrates allows an explicit representation not only of the control regime that is to be employed during a proof, but also of the object-level inference rules that are to be used, i.e. of both I and Σ. Although other systems also allow the explicit representation of proof strategies, not many systems explicitly represent the

inference rules used in an object-level proof. Most of the logic based systems in the literature are Prolog based, and just inherit the inference rules used by the Prolog interpreter for both their object-level and meta-level proofs.

In the previous chapter we identified the essential components of a meta-level inference system as a tuple $(O, I, \Sigma_g, \Sigma_d, \Sigma_t, T)$. We can now illustrate how each of these components is present in the Socrates architecture:

- The object-level theory O is represented in the form of expressions in the logical language as declared by the knowledge engineer, and is accessible at the meta-level through the predicate `object-level-axiom` described above.
- The set of inference-rules I is specified by the knowledge engineer and is represented at run-time in the form of the routine `object-level-inference` described above.
- The search strategy Σ is represented as logical expressions in the meta-level theory. All the subcomponents of Σ (Σ_g, Σ_d and Σ_t), are separately identifiable in this theory, although not explicitly separated in the system.
- The object-level proof tree T is only partially represented in Socrates. At each step in the proof, the meta-level interpreter has access to all possible nodes that can be generated from the current node in the tree (via the routine `object-level-inference`). However, no full representation of the whole proof tree is available. The explicit representation of this tree would enhance the power of Socrates substantially. This limitation is not inherent to the Socrates architecture, but rather a property of the current implementation. It would be a rather simple change to add this feature to the system.

5.4 A discussion of some of the choices made in Socrates

From the above it is clear that certain choices were made in the architecture of Socrates for which alternatives would have been possible. In this section, we will discuss some of these choices, and in particular the way in which they influence the efficiency of the system. Some of the subsections below will justify certain choices made in Socrates, other subsections will argue that, with full hindsight, some things maybe should have been done differently, while yet other subsections discuss options for which we are unable to give a full answer.

5.4.1 The meta-level language

The choice of the language used to describe control regimes at the meta-level is problematic. Currently Socrates employs a logical language for this purpose, but the declarative reading of the expressions in this language in the context of control is far from clear. More often than not the formulae have a very strong procedural flavour. In the example of figure 5.1, axiom [3] has a clear declarative reading, but the other axioms should really be

read not as logical implications (although their surface syntactic structure suggests this), but as the division of goals into an ordered sequence of subgoals. This gives these axioms a very strong procedural reading. This suggests that a logical language might not be the appropriate choice for a meta-level description language in an architecture such as Socrates, although such a language will be useful when it comes to combining or learning control knowledge. ML, originally designed as the meta-level language for describing proof-strategies for LCF ([Gordon et al., 1979] and mentioned in section 2.3.2), is a functional language, whereas other systems in the literature, some of which were described in chapter 2, employ production rule languages, object-oriented languages or block-structured languages.

However, whatever the choice of language for the meta-level would be, it would not affect the main philosophy of meta-level inference, namely that inference activity takes places at the meta-level, thereby simulating object-level proofs as a side effect and it would therefore not have a great bearing on the performance of the overall architecture.

5.4.2 Lazy or eager evaluation of the object-level interpreter

One of the main channels of communication between the object-level and the meta-level is the procedure `object-level-inference` described above. This procedure generates a local section of the object-level search space, providing it to the meta-level for inspection and manipulation. Currently, the procedure eagerly generates all possible descendents of a node in the proof tree. It would be possible to change this behaviour into a lazy generation of nodes on demand. For certain search strategies this would be cheaper (consider for instance a depth-first, non-exhaustive search), and would reduce the costs of both the object-level interpreter itself, its communication with the meta-level, and possibly reduce some of the costs of the meta-level effort as well. It has to be noted however that for many search strategies the meta-level will want to inspect all descendants of an object-level node anyway (for instance in exhaustive strategies, or in best-first strategies). This would completely neutralise the possible gains made by the lazy evaluation of the object-level interpreter. Furthermore, in the context of the current implementation of Socrates, the cost of the eager behaviour of the predicate `object-level-inference` is only a fraction of the total run time of the system.

5.4.3 Provision of default declarations

One of the main reasons for the power and flexibility of the Socrates architecture is the fact that it makes very few assumptions about what the user of the system wants to do. This means that users can mould the system into most of the shapes they require. The other side of this coin is of course that the users have to instruct the system about every aspect of the task they want done, since the system does not make any assumptions on its own. Apart from the fact that this can be quite difficult, it means that very few parts of the Socrates architecture are hardwired. Almost all components of the system are parameterised over declarations of the user. Such components will always be less efficient than modules which have a particular behaviour hardwired into them. One possible compromise would be to supply a number of default modules in the system, for which an efficient implementation

could be provided. Users are then allowed to override such default behaviour, but they would have to pay a price in efficiency for doing so.

For instance, it would be possible to hardwire the declarations for first order predicate calculus as a representation language into the system, with the usual rules regarding commutativity and associativity. It would then be possible to construct a more efficient storage and retrieval mechanism for this particular object-level language.

5.4.4 The naming relation between meta-level and object-level

As mentioned above, formulae at the object-level correspond to constants at the meta-level. The only logical constraint that the naming relation between object-level and meta-level should obey, in general, is that an object-level formula should correspond to a variable free term at the meta-level. The fact that Socrates uses simple atomic constants for these variable free terms, and not more complex terms, is an implementation decision independent of the general architectural structure of the system. However, it could be argued that a richer meta-level representation of object-level formulae than just atomic constants has advantages. In particular, if a meta-level control strategy wants to compute certain properties of an object-level formula, Socrates' use of constants enforces the need to dereference the meta-level name into the corresponding representation at the implementation level, compute the properties, and return the results to the meta-level. All this must be implemented with evaluable predicates that allow the dereferencing of the meta-level name to the implementation language of the system, since this dereferencing is not possible using just the meta-level language. (After all this language does not contain terms that can describe the object-level formula, other than the atomic constant that serves as the name). Had formulae been represented as more complicated terms, then certain properties of the object-level formula could have been computed using the meta-level representation of the object-level formula, rather than having to dereference this representation into the underlying implementation representation.

As an example, consider the formula $F = \forall x \forall y \exists z f(x, y, z)$. Under the current naming relation the meta-level name of this formula is represented as the single meta-level constant $C =$ "$\forall x \forall y \exists z f(x, y, z)$". If a meta-level control strategy wants to compute the number of universally quantified variables in F, the only way to do this is to build an evaluable predicate that dereferences C to F, (or more accurately, to the internal system representation of F), counts the number of universal variables in F, and reports this number back to the meta-level. However, had we used a richer representation for F at the meta-level, say `C' = all(var(1), all(var(2), exists(var(3), f(var(1), var(2), var(3)))))`, where object-level quantifiers, predicates and variables are all represented by a construction of meta-level function symbols and constants, then this computation could have been performed (and more importantly, explicitly specified) at the meta-level itself, rather than having to resort to the lower level of evaluable predicates implemented in Lisp. The terms "quotation-mark names" and "structural-descriptive names" have been used by Tarski [Tarski, 1956] to distinguish these two different approaches to the naming relation between object-level and meta-level.

5.4.5 The representation of object-level failure

The specification of a control regime as given in figure 5.1 is incomplete, since we have to know what the procedural interpretation of this specification is in order to understand which control regime it specifies. This meta-level control regime is used in an essential way: the way in which the object-level backtracks between conjuncts in a proof is directly related to the way in which the meta-level backtracks between the conjuncts of clause [4]. The backtracking behaviour of the object-level is not explicitly specified at the meta-level at all, but is implicitly encoded using knowledge about the backtracking behaviour of the meta-level. Furthermore, the meta-level proof procedure fails if it fails to find an object-level proof. It can be argued that it would be better if the meta-level proof procedure does not fail in such a case, but that it succeeds, returning a special status indicating that the object-level proof has failed. Thus, object-level backtracking should be explicitly specified at the meta-level, rather than being implicit in the meta-level backtracking, and object-level failure should be explicitly represented as well, rather than being represented by meta-level failure.

This is an example of how the power of a meta-level inference is "model relative", as discussed in the previous chapter: the power of the Socrates meta-level is limited by the power of the model of the object-level computation that is available to it.

5.4.6 Logical soundness

A problem with the system in general and with the meta-level interpreter in particular is the issue of logical soundness. As already discussed above, the current system allows the user to declare arbitrary sets of inference rules for the propositional part of the object-level logic (the quantificational part of the proof theory is hardwired in the retrieval mechanism). This allows the user to declare potentially unsound sets of inference rules, an obviously undesirable situation. As described in section 5.1.1 above, it is possible to the restrict the declaration of the proof theory to a process of selection of a subset from a set of sound inference rules, thereby making it impossible to introduce unsoundness in the proof theory. However, this approach breaks down when the user introduces new logical connectives that are not included in the library of predefined sets of inference rules. A further source of unsoundness is the meta-level interpreter. Not only do the control strategies at the meta-level affect the completeness of the system (as they are intended to do in order to cut down the search space), but unfortunately they can also affect the soundness. Taking the example of figure 5.1, we can change clause [4] into:

```
[4']   (∀ F, Lhs, Rhs, S, LhsSubst, RhsSubst)
       [compound-expression(F) ∧
        split-compound-expression(F, Lhs, Rhs) ∧
        proof(Lhs, S) ∧
        → proof(F, S)
       ]
```

and thereby reducing the proof of a compound expression to the proof of the left-hand side of that expression only. This would obviously produce an unsound system, but the user is

in no way prevented from making mistakes like this. Again, we could introduce a library system, and allow the user only predefined strategies from this library, but this would severely restrict the flexibility and power of the system. Other approaches are possible, such as the one in the LCF/ML system [Gordon et al., 1979], where a type system enforces the soundness of the system, as described in section 2.3.2. (See also chapter 8 where another solution to this problem is proposed).

Part II

Measuring and improving the performance of meta-level inference systems

Chapter 6

The problem of meta-level overhead

In chapter 3 we argued in favour of a particular type of meta-level architecture, the bilingual meta-level inference systems, and in chapter 5 we showed how such an architecture can be built. However, as we will discuss in this chapter, this type of system also suffers from a serious problem, related to the run time efficiency of such a system. In the previous chapters we have argued in favour of meta-level architectures on the basis of their ability (among other things) to cut down the object-level search space. However, two potential problems hamper the performance of meta-level systems. These two problems, to be discussed in more detail in this chapter, are briefly speaking as follows.

The first problem has to do with the cost/benefit structure of meta-level reasoning. On the one hand a meta-level architecture gives us increased flexibility of the behaviour of the system, allowing us to formulate efficient search strategies. On the other hand a meta-level architecture has to interpret at run time the explicitly represented strategy, instead of executing a hardwired, implicit control strategy, as a single-level system would do. This gives rise to a situation of cost and benefit. The benefit is the reduced search space achieved through the formulation of explicit control strategies, the cost is the extra meta-level computation to be performed by the system. As a result, part of the gain of a meta-level architecture is lost through the costs of the explicit interpretation. In the first part of this chapter we will present a theoretical model of this trade-off which will show that in general there will indeed be a point beyond which the meta-level investment of a system outweighs the object-level savings. Thus, the trade-off is between a reduction in the number of object-level inferences that the system has to make versus an increase in the cost per single object-level inference. Obviously, a meta-level inference system will only be of realistic use if the increase in cost per object-level inference is outweighed by the reduction in the number of object-level steps. This problem is well acknowledged in the literature (although not often quantitatively investigated), and applies to all meta-level architectures. As we shall see in the second part of this chapter, this problem is of particular significance for bilingual meta-level inference systems. In this case, the cost of maintaining the naming relation between the object-level and the meta-level language aggravates the increase in cost per object-level inference, and thus requires a particularly large reduction in the number of object-level steps to offset this effect.

We will call this problem the problem of *meta-level overhead.* The purpose of this chapter is to analyse the cause and size of this problem in more detail. We will find that

the increase in cost per object-level inference is significant and especially so for bilingual systems. This will lead us to the question we will try to solve in the remaining chapters: how can we reduce the meta-level overhead without losing the advantages of an explicit meta-level?

6.1 Costs versus savings of meta-level inference

In the context of meta-level overhead it is important to distinguish between two different quantities, on the one hand the number of steps necessary to solve a problem, and on the other hand the cost of one of these steps. What meta-level inference gives us is enough flexibility to specialise the architecture in such a way that the number of steps in the (object-level) search can be greatly reduced. The problem lies in the fact that this flexibility comes at the expense of an increase in the cost for each single object-level step, because a number of meta-level steps might be needed for any single object-level step. For meta-level inference to work, it is crucial that this increase in the cost of a single simulated object-level step does not outweigh the reduction in the number of such steps. In this section we will present a simple model of a meta-level architecture (based on a model given in [Rosenschein and Singh, 1983]). This model will show that in general there is a point beyond which the meta-level investment of a system outweighs the object-level savings.

In our model we will assume that a system has two independent methods for solving a particular object-level problem, although the model can be easily extended to an arbitrary number of methods. It is assumed that without meta-level reasoning, the system will try to use the two methods in a random order to solve the problem. Again, the model can be easily adjusted to accommodate for the more realistic assumption that the system will execute the methods in some fixed order if no meta-level reasoning is done. Some remarks about such an extension are made at the end of this section. The goal of any meta-level effort in the system is to find the optimal order for the two methods, resulting in a saving of object-level inference at the expense of some meta-level inference. We will use the model to investigate the trade-off between these two.

Let us call the two object-level methods available to the system for solving the problem x and y. We will assume that each of these methods has a certain probability of solving the problem, say p_x and p_y, initially unknown to the system. Furthermore, each method has some associated expected cost of executing that method, say c_x and c_y. Again, these expected costs are initially unknown to the system. In our model we will assume that c_x is independent of whether x succeeds or not (and similarly for c_y). This assumption can be relaxed to distinguish between c_x^f, the cost of x when x fails and c_x^s, the costs of x when x succeeds (see the remarks at the end of this section). Given these assumptions about x, y, c_x, c_y, p_x and p_y, the expected cost of executing x before y, $exec([x;y])$ is

$$exec([x;y]) = c_x + (1 - p_x)c_y \tag{6.1}$$

namely the expected cost of executing x plus the expected cost of executing y, but reduced by the chance that y is not executed because x has succeeded in solving the problem. An analogous expression holds for $exec([y;x])$. The decision to try x before y should be made when

$$exec([x;y]) < exec([y;x])$$

or equivalently, using (6.1):

$$p_y/c_y < p_x/c_x.$$

The quantity $\phi(z) = p_z/c_z$ (for $z = x, y$) can be seen as a measure of the utility of a method z. The above inequality says that the method with the highest utility should be tried first.

As mentioned above, the values for success rates and the expected costs of x and y (and therefore the values of $\phi(x)$ and $\phi(y)$) will in general not be available to the system, and will have to be computed at the cost of some meta-level effort, say c_m. Once the meta-level has computed $\phi(x)$ and $\phi(y)$, the optimal ordering for the two methods is known. We can now derive the savings s made by executing the methods in this optimal order as follows: assume that without any meta-level effort, the system chooses a random ordering of x and y. The savings are then the cost of executing a randomly chosen method minus the cost of executing the methods in the optimal ordering, increased with the cost of finding the optimal ordering:

$$\text{savings} = \text{cost-of-random-choice} - (\text{cost-of-optimal-choice} + \text{meta-level-cost})$$

The expected cost of executing the methods in a random ordering is

$$\frac{exec([x;y]) + exec([y;x])}{2}.$$

If the system spends c_m on meta-level effort and then chooses x before y as the optimal ordering, the execution cost would be:

$$exec([x;y]) + c_m$$

The saving s would then be the difference between these two formulae:

$$\frac{exec([y;x]) - exec([x;y])}{2} - c_m$$

This would be the saving if the system preferred x over y. Making this argument symmetrical in x and y, we get as the expected savings s:

$$s = \frac{|exec([x;y]) - exec([y,x])|}{2} - c_m = \frac{|p_x c_y - p_y c_x|}{2} - c_m. \tag{6.2}$$

In general, we cannot expect that the meta-level will always succeed in computing the true values of $\phi(x)$ and $\phi(y)$. We can adjust our model to the assumption that the meta-level prefers x over y (i.e. it claims $\phi(x) > \phi(y)$), but that this decision is only correct with a probability p_m. In this case the expected savings of preferring x over y are

$$s([x;y]) = p_m(p_x c_y - p_y c_x) + (1 - p_m)(p_y c_x - p_x c_y) - c_m,$$

namely: the expected gain for the correct decision ($exec([x;y]) - exec([y;x]) = p_x c_y - p_y c_x$) times the probability that it was indeed correct, plus the expected gain when the decision was incorrect (or actually the loss, since $p_y c_x - p_x c_y$ is a negative quantity because we assumed $\phi(x) > \phi(y)$) times the probability the decision was incorrect, minus the meta-level cost c_m. The above expression can be simplified to

$$s([x;y]) = (2p_m - 1)(p_x c_y - p_y c_x) - c_m$$

These are the expected savings when the meta-level reasoning prefers x over y. An analogous expression holds for $s([y;x])$. Combining these two, and again taking into account that without meta-level effort the system randomly orders x and y, therefore getting it right half the time, the expected savings of meta-level reasoning are

$$s = \frac{(2p_m - 1)\Delta_{x,y}}{2} - c_m \tag{6.3}$$

where $\Delta_{x,y}$ is a notation for $|p_x c_y - p_y c_x|$. Notice that when $p_m = 1$, formula (6.3) reduces to formula (6.2) above.

In realistic situations, the value of p_m, the probability of making a correct choice between methods, will be dependent on the amount of meta-level effort spent, i.e. $p_m = f(c_m)$. Placing this in equation (6.3) above, we get

$$s(c_m) = \frac{(2f(c_m) - 1)\Delta_{x,y}}{2} - c_m \tag{6.4}$$

Obviously, we want to maximise s as a function of c_m. Exactly what shape $s(c_m)$ will have will depend on how the accuracy of the meta-level reasoning depends on the meta-level effort spent by the system, i.e. $f(c_m)$. The following are reasonable assumptions to make about $f(c_m)$:

- We expect of a meta-level that the quality of computations does not decrease with increased effort, making $f(c_m)$ non-decreasing:

$$\frac{df}{dc_m}(c_m) \geq 0 \tag{6.5}$$

- With no meta-level effort, i.e. $c_m = 0$, our model assumes $p_m = \frac{1}{2}$ (because of the random ordering of x and y):

$$f(0) = \tfrac{1}{2} \tag{6.6}$$

  Notice that $f(0) = \frac{1}{2}$ implies $s(0) = 0$, which is the correct boundary condition for s.

- Since $f(c_m)$ is a probability, we certainly expect $0 \leq f(c_m) \leq 1$. Together with (6.5) and (6.6) above this gives:

$$\tfrac{1}{2} \leq f(c_m) \leq 1 \tag{6.7}$$

- Finally, although this is not strictly necessary, we can expect some effect of diminishing returns, giving a smaller increase of meta-level correctness for every further increase in c_m:

$$\frac{d^2 f}{dc_m^2}(c_m) \leq 0 \tag{6.8}$$

For illustrational purposes it is interesting to look at a number of example functions for $f(c_m)$ which have the above properties, to see what the actual shape of the savings curve would be for these functions.

Figure 6.1a shows the case for $p_m = f(c_m) = 1 - \frac{1}{2}e^{-c_m}$. It is important to stress that the only significance of this function is that it obeys conditions (6.5)-(6.8), but otherwise

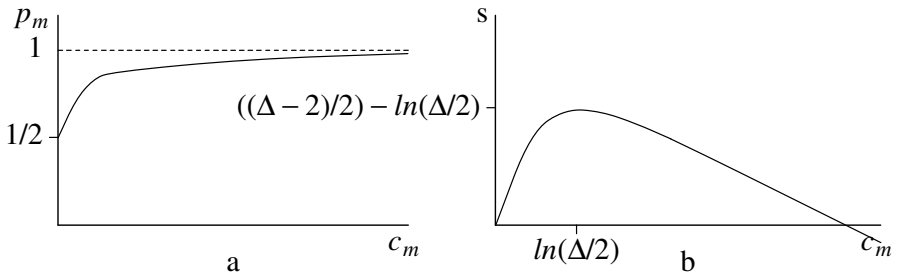


Figure 6.1: first example of a meta-level cost function

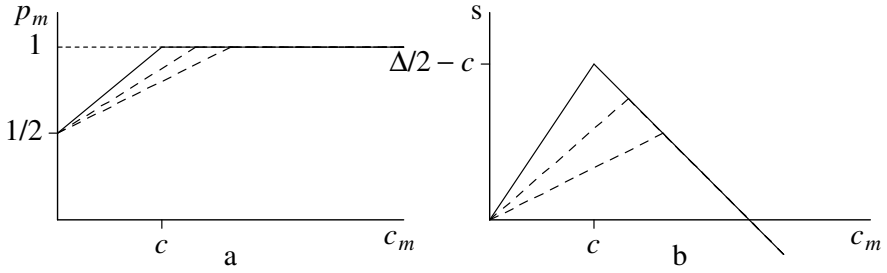


Figure 6.2: second example of a meta-level cost function

it is arbitrary. It serves to illustrate the possible case where a relatively small amount of meta-level effort results in substantially improved performance, while there are diminishing returns for subsequent effort. Its convergence to $p_m = 1$ for $c_m \to \infty$ is arbitrary, and does not influence the qualitative shape of the savings curve for $s(c_m)$ shown in figure 6.1b. In this figure we can see that at a certain point, the savings achieved by meta-level reasoning reach a maximum, and any further meta-level effort will only reduce the overall savings. With even more effort spent on meta-level reasoning, the system will eventually behave worse than without any meta-level effort at all. Similarly, figure 6.2a shows an example where increased meta-level effort initially pays off, but does not contribute to a more effective control decision beyond a certain threshold c. The function used in this figure (shown as a solid line) is $p_m = f(c_m) = \frac{1}{2}c_m/c + \frac{1}{2}$ on the interval $[0, c)$, and $p_m = f(c_m) = 1$ on $[c, \infty)$. Again, the only crucial properties of this example for $f(c_m)$ are conditions (6.5)-(6.8). Other properties, such as the slope of $f(c_m)$ on $[0, c)$, or the fact that $f(c_m) = 1$ on $[c, \infty)$ are irrelevant to the qualitative shape of the saving curve shown as the solid line in figure 6.2b, where again the savings reach a maximum at some point, beyond which further meta-level effort will only degrade the performance of the system.

Figure 6.2 also shows the changing behaviour of the system when it tries to solve harder meta-level problems. We call a meta-level problem *harder* if for the same amount of meta-level effort c_m, the system achieves a lower value of p_m (i.e. the choice between the applicable methods is made less reliably). In the case of the definition for $f(c_m)$ used in figure 6.2, this means an increasing value of c as indicated by the family of dashed lines in figure 6.2a. The corresponding behaviour of the saving function $s(c_m)$ is shown

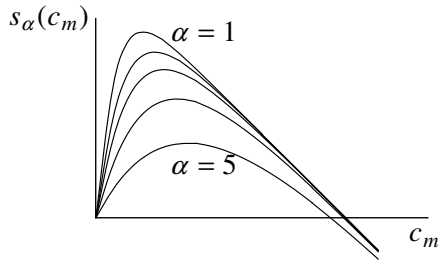


Figure 6.3: savings functions for harder meta-level problems

by the dashed lines in figure 6.2b. We see that if the meta-level problem gets harder, the optimum meta-level effort is found for a larger value of c_m, and the corresponding savings are reduced. This behaviour illustrates a phenomenon often observed in developing meta-level systems, namely that the usefulness of meta-level inference cannot be illustrated adequately on very simple toy problems. For those problems, c will be very small, and any significant amount of meta-level effort is likely to be larger than c, and will thus overshoot the point of maximum utility. A similar set of curves can be drawn for the example used in figure 6.1. If we take $f(c_m) = 1 - \frac{1}{2}e^{-c_m/\alpha}$ (instead of $f(c_m) = 1 - \frac{1}{2}e^{-c_m}$), then increasing values of $\alpha \geq 1$ will represent harder meta-level problems. Again, the maxima for $s(c_m)$ for different values of α lie at increasing values of c_m. An example of this behaviour is shown in figure 6.3 which displays different curves $s(c_m)$ for values $\alpha = 1,\ 1.5,\ 2,\ 3,\ 5$.

Looking back at the simple model described above, we can see that there are some assumptions which can be relaxed without significantly changing the conclusions we draw from the model.

- number of methods:

  Rather than modelling just two methods x and y, we can adjust to model to choose between n methods. Expression (6.1) can be generalised from 2 to n methods, so that the cost of executing methods $x_1, \ldots, x_n$ in that order is

  $$exec([x_1; \ldots; x_n]) = c_1 + \sum_{i=2}^{n}(\prod_{j=1}^{i-1}(1 - p_j))c_i \tag{6.9}$$

  (i.e. the total cost is the sum of the cost of each method multiplied with the chance that all earlier methods failed). The optimal order for executing the methods would of course be some order $x_{i_1}; \ldots; x_{i_n}$ such that for all $j, 1 \leq j \leq n-1, \phi(x_{i_j}) \geq \phi(x_{i_{j+1}})$.

- initial ordering of methods:

  The model assumed that with no meta-level effort the system would make an arbitrary choice for the order in which it executed its object-level methods x and y. A more realistic assumption would be that the system would apply x and y in some fixed order, say first x and then y, on the basis of some a priori knowledge that the system's designer has about $\phi(x)$ and $\phi(y)$. Suppose that with no meta-level effort, the system chooses x before y, and that this choice is indeed the right one with a chance p_0. In

other words, the value $f(0)$, the quality of the meta-level decision at no meta-level effort, is no longer $\frac{1}{2}$, as specified in (6.6), but is now p_0. Presumably, $\frac{1}{2} < p_0 \leq 1$, since the fixed ordering programmed into the system will be better than a random ordering. Because (6.6) has changed to

$$f(0) = p_0 \tag{6.10}$$

formula (6.7) must also change, into

$$p_0 \leq f(c_m) \leq 1 \tag{6.11}$$

We would like our new version (6.10) to imply $s(0) = 0$, just as (6.6) did, so we have to recalibrate (6.4). Furthermore, rather than multiplying $(2f(c_m) - 1)\Delta_{x,y}$ by $\frac{1}{2}$, as in formula (6.4), representing the initial random ordering, we should multiply with $(1 - p_0)$, the chance that the a priori decision would have been wrong. As a result of these changes, and after algebraic simplifications, the new version of the savings equation (6.4) becomes

$$s(c_m) = 2(1 - p_0)(f(c_m) - p_0)\Delta_{x,y} - c_m \tag{6.12}$$

Notice that (6.12) is equal to (6.4) for the case $p_0 = \frac{1}{2}$ (the random choice case). Furthermore, given an extra assumption about the behaviour of $f(c_m)$ we can prove that the value of (6.12) is always smaller than the value of (6.4). This is just what is to be expected, since with the fixed ordering the system performs better without any meta-level effort than it did before without meta-level effort, and as a result the potential savings that can be made by the meta-level are smaller. The additional assumption about $f(c_m)$ that is needed to ensure that (6.12) $\leq$ (6.4) is that, when p_0 increases, $f(c_m)$ does not increase more than p_0 for any value of c_m. Formally, if $f_{p'}$ and $f_{p''}$ are two versions of $f(c_m)$ for different values of $p_0 = p', p''$, with $p'' > p'$, we require:

$$\forall c_m : f_{p''}(c_m) - f_{p'}(c_m) \leq p'' - p'$$

This is a reasonable assumption to make, since we cannot expect that the a priori knowledge about the relative utilities of x and y will increase the quality of the meta-level effort by more than just this a priori amount. The proof goes as follows: for $p_0 = \frac{1}{2}$ we have (6.12)=(6.4). If p_0 increases, the value of (6.12) decreases, since the value of $(1 - p_0)$ decreases, and the value of $(f(c_m) - p_0)$ also decreases. This last statement is only true because of our extra condition: generally, if p_0 increases, $f(c_m)$ will increase, but our extra condition states that it will not increase more than p_0, so that $(f(c_m) - p_0)$ either stays the same or decreases as well, so that the whole value of (6.12) decreases, making it less than (6.4) for any value of $p_0 > \frac{1}{2}$.

- cost independent of success:

  The assumption above was that a method z had some associated expected cost c_z which was independent of whether the method succeeded or failed. This assumption can be lifted by introducing for any method z two costs, namely c_z^s, the cost of z if z succeeds, and c_z^f, the cost of z if z fails. The expected cost of executing a method z

is then $c_z = p_z c_z^s + (1 - p_z)c_z^f$, namely the cost of z succeeding times the probability it will succeed plus the cost of z failing times the probability it will fail. We can then uniformly substitute this new expression for any c_z in the above, for $z = x, y$. If we do this in expression (6.1), representing the cost of executing first method x and then method y, the resulting expression can be rewritten to the following:

$$exec([x; y]) = p_x c_x^s + (1 - p_x)(c_x^f + p_y c_y^s + (1 - p_y)c_y^f) \tag{6.13}$$

which is exactly what we expect, namely the cost of x succeeding times the probability that x succeeds plus the sum of x failing and executing y times the probability that x fails.

6.2 The cost of meta-level inference

The previous section showed that there will indeed be a point beyond which increased meta-level effort will outweigh the benefits gained at the object-level. It therefore becomes a matter of considerable practical concern to find out where this trade-off point lies in a particular system. If the increase in cost per object-level step due to meta-level effort is significant, then it is less likely that this increase will be outweighed by the reduction in the number of object-level steps needed for a particular task.

We can expect this increase in cost per object-level step to be higher for bilingual than for mono-lingual systems, because bilingual systems need the extra effort of translating between object-level formulae and their meta-level names. In order to assess the size of this problem we will compare both a mono-lingual and a bilingual meta-level system (to be called I_1 and I_2) with some single level system (I_0) that has a hardwired control strategy. We will measure the performance of each of these interpreters for the same task, using the same control regime, although explicitly represented in I_2 and I_1 and hardwired in I_0. We will expect I_0 to outperform the other two, since all three interpreters run the same control regime, but I_1 and I_2 will have to explicitly interpret the control regime. What we are interested in is the difference in overhead between I_1 and I_2 relative to I_0, which will be an indication of the costs of I_2 being bilingual. We are not so much interested in the performance difference between I_0 on the one hand and I_1 and I_2 on the other. This comparison would in some sense be unfair, since for the purpose of the experiment we restrict all three interpreters to run the same control regime. However, I_1 and I_2 have the potential to run other (possibly more efficient) control regimes whereas the behaviour of I_0 is in this respect fixed. Thus, the fact that I_0 outperforms both I_1 and I_2 should not be taken as an argument against explicit meta-levels, since the behaviour of I_1 and I_2 could potentially be adjusted to a control regime more appropriate for the particular task, leading to increased efficiency.

In order to do the comparison between a mono-lingual, a bilingual and a hardwired interpreter, we need an environment in which we can build all these without too many irrelevant differences between them that might distort our results. We cannot use the Socrates environment for the experiment, since it is by design only bilingual. Instead, we will use a Prolog system as the environment for our experiments. Prolog is often claimed to be particularly well suited for building interpreters, and the Prolog literature contains

```
solve(true).
solve((LeftGoal, RightGoal) :- solve(LeftGoal), solve(RightGoal).
solve(Goal) :- clause((Goal:-Body)), solve(Body).
```

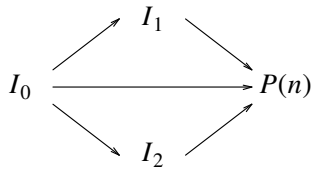
figure a

```
solve([], []).
solve([G|Gs], S) :-
    get_clause(G, C),
    rename_vars(C, [G|Gs], C1),
    head(C1, H1),
    unify(G, H1, S1),
    body(C1, B1),
    append(B1, Gs, NewGs),
    instantiate(NewGs, S1, NewGsPlusS1),
    solve(NewGsPlusS1, S2),
    compose(S1, S2, S).
```

figure b

Figure 6.4: code of meta-circular interpreters in Prolog

descriptions of both mono-lingual and bilingual meta-interpreters ([Kowalski, 1979], [Sterling and Shapiro, 1986], [O'Keefe, 1985]). Furthermore, since these meta-interpreters are often meta-circular (in the sense discussed in section 2.4.3), their behaviour is functionally equivalent to that of the Prolog base level system in which they are implemented. Thus, we can use the Prolog base level system as our hardwired interpreter I_0. We can represent our experimentation environment schematically as:



where $P(n)$ stands for some object-level program with input of length n that is to be executed by either the base-level interpreter I_0 or by one of the two meta-interpreters I_1 of I_2, which are themselves, of course, in turn executed by I_0.

The code for the interpreters I_1 and I_2 is given in figure 6.4. Figure 6.4a lists the code for the most simple version of a meta-circular interpreter for pure Prolog. The meta-level interpreter represents clauses in the same way as the base level interpreter does, with the `,`-functor used to represent conjunctions as usual. This interpreter is mono-lingual, since Prolog is used to represent both the meta-interpreter and the object-level program it interprets.

Figure 6.4b lists the top-level code for a bilingual meta-interpreter for Prolog. Further explicit definitions of all the predicates `get_clause/2, rename_vars/3, head/2, unify/3, body/2, instantiate/3` and `compose/3` must obviously also be provided. This meta-interpreter is bilingual, since the language used to represent the object-level program is different from the language in which the meta-interpreter itself is written (Prolog). Object-level clauses are represented as Prolog clauses except for the variable representation. The object-level variables are represented as `var(x), var(y), ...` instead of the Prolog variables `X, Y, ...` Thus, an object-level literal like `p(X)` for I_0 and I_1 would look like `p(var(x))` for I_2. The predicates used in the code for I_2 (like `unify/3` etc.) are programmed to deal with this variable representation, rather than with the standard Prolog variables. This means that I_2 must explicitly manipulate substitutions for object-level variables, rather than implicitly use the mechanism provided by the base-level interpreter, as I_1 does. As a result, I_2 is a 2-place predicate instead of the 1-place predicate for I_1[1].

I_2 also has a slightly different representation of object-level conjunctions, representing them as lists instead of using the `,`-functor, but this difference will not influence any measurements, since neither the `,`-functor nor the list-constructor has any special status in the Prolog systems that we will use. Strictly speaking, the code for I_2 from figure 6.4b is not quite correct, since it does not explicitly represent the object-level theory. This can easily be changed, by adding a third argument to `solve/2`, consisting of a list of clauses of the object-level theory. This argument would then be passed as an additional argument into `get_clause/3`. However, the implementation shown in figure 6.4b allows us still to use the hardwired indexing mechanism of the Prolog system for the object-level theory. Due to the lack of separation in the Prolog language between function-symbols and predicate-symbols, the predicate `get-clause/2` can still use the built-in database indexing to access the object-level predicates (which are strictly speaking function symbols at the meta-level). This programming trick removes one difference between I_2 on the one hand and I_1 and I_0 on the other which might otherwise distort the measurements that we are interested in. A final remark to be made about the code for I_2 in figure 6.4b is that it performs standard depth-first Horn Clause resolution, exactly as the code for I_1 in figure 6.4a although encoded slightly differently. A version of I_1 that encodes depth-first search in the same way as I_2 and that also uses lists to represent conjunctions would look like:

```
solve([]).
solve([G|Gs]) :-
    clause(G, B),
    append(B, Gs, NewGs),
    solve(NewGs).
```

The differences between this version of I_1 and the version of figure 6.4a will not influence our measurements, and because the code in figure 6.4a will make some of the arguments later in this section somewhat easier to present we will use the version of I_1 as given there.

Having described the purpose and setting of the experiment, we have to decide how to measure the performance of the three interpreters. Although many objections can

[1]The reader should remember that the advantage of this additional complexity of I_2 is that the object-level language for I_2 can be changed, whereas the object-level language of I_1 must always be the same as the meta-level language, namely Prolog.

be made against it (see below), we will use the standard performance measure for logic programming systems, namely the number of Logical Inferences. The number of logical inferences LI performed by a Prolog goal P, by executing the clause $P : -P_1, \ldots, P_n$ is defined as

$$\mathrm{LI}(P) = 1 + \sum_{i=1}^{n} \mathrm{LI}(P_i)$$

If P is a unit clause (i.e. $n = 0$), then P performs 1 logical inference. To illustrate this definition, we take the standard code for the predicate `member/2`:

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

An example call to this predicate, such as

```
member(3,[1,2,3,4]).
```

would make 3 logical inferences, namely

```
LI(member(3,[1,2,3,4]))         =
    1 + LI(member(3,[2,3,4]))   =
    1 + 1 + LI(member(3,[3,4])) =
    1 + 1 + 1                   = 3
```

As mentioned, a number of objections can be made against the logical inference as a performance measure:

The first objection is one that does not actually affect the use of the LI-measure for our purposes, but reduces its usefulness in other contexts. The definition of logical inference takes into account only the clauses that are executed as part of the proof tree. Many other clauses might have been executed in the search space of the interpreter, but only the successful clauses that contribute to the solution are counted. This can again be illustrated using the example call to `member/2` above. In the computation of `member(3, [1,2,3,4])`, no account was taken of the fact that the base clause failed for all but the last recursive call, although, presumably, the Prolog system had to spent some effort trying to match the head of this unit clause with the goal and failing. Even worse, if our base clause had read:

```
member(X, [Y|_]) :- expensive_calculation, X=Y.
```

the LI measure would not reflect the effort spent in performing `expensive_calculation` twice as part of the failing base clause. In other words, the LI-measure takes account only of the size of the proof tree, and not of the size of the search tree (where the proof tree is only a subset of the search tree). For our experiment this property of the LI-measure is not important since its effects are equal for all three interpreters. This is because all three interpreters run the same control strategy, and therefore traverse the same search tree to find the same proof tree, and thus the LI-measure blurs the distinction between search tree and proof tree in equal amounts for all of the interpreters. Since, we are only interested in the performance of each interpreter relative to the others, and not in any absolute speed measurements, this objection to the LI-measurement will not influence our results. Nevertheless, this seriously disqualifies the logical inference as a performance measure

between interpreters that run different control regimes: both efficient and inefficient control regimes would score the same LI-rating, since they both eventually find the same proof tree. The only possible way of using the LI-rating in such a case is to take into account the time it took the interpreter to make the logical inferences, in other words, to measure the number of logical inferences an interpreter can make per second, LI/*sec*, often written as LIPS. In that case the efficient control regime will get the same number of logical inferences in less time, thereby scoring a higher LIPS-rating.

The second objection against LI as a performance measure is that it only measures the cost of executing the conjuncts $P_1, \ldots, P_n$, and that it ignores the cost of executing the conjunction itself. This objection is indeed relevant for our experiment, since as we will see below, the cost of executing a conjunction is much higher for I_2 than it is for I_1 and I_0, and is a function of the number of shared variables between the conjuncts. The fact that this cost is ignored in the cost-measure of the task therefore underestimates the cost of I_2 compared to the cost for I_1 and I_0.

The third objection agains LI as a performance measure is that it assumes that each procedure call has the same (unit) cost. Procedure calls are of course not of uniform cost, since different amounts of computation are performed for the unification of the arguments in the head of different procedure calls. In this context the terms *step complexity* and *unification complexity* are sometimes used. The LI-measure only takes account of the step complexity and ignores the unification complexity. Again, this objection is relevant to our experiment since unification is much more expensive in I_2 than in I_0 and I_1.

On the basis of these objections, a more appropriate definition for $\mathrm{LI}(P)$ would be

$$\mathrm{LI}(P) = f(P) + \sum_{i=1}^{n}(\mathrm{LI}(P_i) + g(P_i)) + h(P)$$

where $f(P)$ represents the unification complexity of P, $g(P_i)$ represents the cost of P_i sharing variables with other conjuncts, and $h(P)$ is a measure of the effort spent on the failing or-branches for P. However, since we can give no reasonable estimates for f, g or h, we will cautiously use the conventional measure, while bearing in mind the objections mentioned above. Notice that our LI measure will only underestimate the value of the computational effort (since f, g and h are all positive values), so that our measurements of computational effort will be conservative estimates of the real values. Before we proceed to discuss the measurements we will be taking, we introduce a further notation. For some object-level program P, with input of length n, and some interpreter I_i $(i = 0, 1, 2)$, we will write $\mathrm{LI}(I_i, P(n))$ to mean the number of logical inferences it takes I_i to execute program P on input n, and similarly $\mathrm{LIPS}(I_i, P(n))$ for the corresponding LI/*sec*-rating. For interpreter I_0, the program it executes can not only be some object-level program P, but also some other interpreter I_i $(i = 1, 2)$ which is in turn executing some object-level program P. Using the same notation, the number of logical inferences required by I_0 in this situation would be written as $\mathrm{LI}(I_0, I_i(P(n)))$ $(i = 1, 2)$.

Given the above descriptions of the experimental environment and the performance-measurement, and using the notation introduced above, we can answer our previously defined question (comparing the overheads of both I_1 and I_2 with respect to I_0) in two ways:

1. Given an object-level task P which takes an input of length n, we can compare the overheads in terms of the time it takes for all three interpreters to execute $P(n)$. This corresponds to comparing

$$\frac{\text{LIPS}(I_1, P(n))}{\text{LIPS}(I_0, P(n))} \qquad \text{with} \qquad \frac{\text{LIPS}(I_2, P(n))}{\text{LIPS}(I_0, P(n))}$$

in other words, we are interested in the ratio

$$\frac{\text{LIPS}(I_1, P(n))}{\text{LIPS}(I_2, P(n))}$$

Since P is the same in all cases, and since all three interpreters I_i, $i = 0, 1, 2$ execute the same control strategy, we have

$$\text{LI}(I_0, P(n)) = \text{LI}(I_1, P(n)) = \text{LI}(I_2, P(n))$$

so all we would have to measure here is the time taken by each interpreter to perform $P(n)$.

2. Alternatively, we can compare the number of logical inferences it takes I_0 to execute $P(n)$ either directly, or via I_1 or via I_2. This corresponds to comparing

$$\frac{\text{LI}(I_0, I_1(P(n)))}{\text{LI}(I_0, P(n))} \qquad \text{with} \qquad \frac{\text{LI}(I_0, I_2(P(n)))}{\text{LI}(I_0, P(n))}$$

in other words, we are interested in the ratio

$$\frac{\text{LI}(I_0, I_2(P(n)))}{\text{LI}(I_0, I_1(P(n)))}$$

Division of any of the three quantities $\text{LI}(I_0, I_i(P(n)))$ for i=0,1,2 (where we should read $\text{LI}(I_0, I_0(P(n)))$ as $\text{LI}(I_0, P(n))$) by the time taken for that task should presumably result in the same figure, namely the LIPS rating of I_0.

The first of these two approaches measures the meta-level overhead in terms of the speed of each of the interpreters, and the second approach measures the overhead in terms of the effort needed by the base-level implementation. If our measurements could be done accurately, and if the LI and LIPS were indeed good measures of the run-time complexity of logic programs, then both approaches should result in the same conclusion, in other words, we would expect

$$\frac{\text{LI}(I_0, I_2(P(n)))}{\text{LI}(I_0, I_1(P(n)))} = \frac{\text{LIPS}(I_1, P(n))}{\text{LIPS}(I_2, P(n))} \tag{6.14}$$

However, because of the deficiences in the definition of LI as discussed above, and because of the difficulties in measuring LIPS (explained in detail in section 6.3), we cannot expect this equality to hold in practice. Each of these measurements can of course be done for either a fixed object-level task P, with input of fixed length n, or over a variation of both n and P. If the logical inference is indeed a good measure of complexity, then all of these

Prolog system	I_0	I_1	I_2	I_0/I_1	I_0/I_2	I_1/I_2
Edinburgh	7500	311	0.7	24	10700	445
Quintus	20000	850	2.0	24	10000	425
BIMProlog	17000	326	1.6	52	10600	204

Figure 6.5: LIPS ratings on `nrev(30)`

measurements should lead to the same conclusions about the difference in overhead for I_1 and I_2.

The advantage of the first approach (using the LIPS-ratings), is that it directly measures the speed of the different interpreters, which is the property we are interested in. The disadvantage is that it involves measuring execution times of programs, which are always prone to variations due to unpredictable effects in operating system and hardware. The advantage of the second approach (using the LI-ratings) is that these ratings can be computed or measured exactly, rather than only approximately. The disadvantage of this approach is that these LI-ratings are only of indirect interest. In themselves they do not tell us anything about the speeds of the different interpreters, and only relate the speeds of I_1 and I_2 to the speed of I_0 via the following equation:

$$\mathrm{LIPS}(I_i, P(n)) = \mathrm{LIPS}(I_0, P(n)) \times \frac{\mathrm{LI}(I_i, P(n))}{\mathrm{LI}(I_0, I_i(P(n)))} \quad \text{for} \quad i = 1, 2 \tag{6.15}$$

The motivation for this equation is as follows: the speed of I_i (in logical inferences per second) equals the speed of I_0 divided by the number of logical inferences I_0 needs to make for each logical inference of I_i. This number of logical inferences I_0 needs to make for each logical inference of I_i can be represented as the quotient of the total number of steps I_0 makes for executing a task via I_i and the total number of steps I_i has to make to that task, ie $\mathrm{LI}(I_0, I_i(P(n)))/\mathrm{LI}(I_i, P(n))$. Dividing the speed of I_0, $\mathrm{LIPS}(I_0, P(n))$, by this quotient gives the above relation between the speed of I_i and the speed of I_0. Notice that the above equation is trivial for $i = 0$ (if we read $\mathrm{LI}(I_0, I_0(P(n)))$ as $\mathrm{LI}(I_0, P(n))$).

Equation (6.15) can also be used to derive equation (6.14) above, which stated that both approaches to measuring the ratio of overhead of I_1 and I_2 (using either LI or LIPS ratings) should give the same result. Taking the right-hand side of (6.14), and applying equation (6.15) to both numerator and denominator, for I_i, $i = 1, 2$ respectively, we obtain

$$\frac{\mathrm{LIPS}(I_0, P(n)) \times \frac{\mathrm{LI}_{(I_1, P(n))}}{\mathrm{LI}_{(I_0, I_1(P(n)))}}}{\mathrm{LIPS}(I_0, P(n)) \times \frac{\mathrm{LI}_{(I_2, P(n))}}{\mathrm{LI}_{(I_0, I_2(P(n)))}}}.$$

After applying the equality $\mathrm{LI}(I_1, P(n)) = \mathrm{LI}(I_2, P(n))$ and algebraic simplification we obtain the left-hand side of equation (6.14) as desired.

After all these preliminary discussions we are finally ready to describe the experiments that embody the two ways discussed above of measuring the meta-level overhead. Our first experiment measures the first of these two, the $\mathrm{LIPS}(I_i, P(n))$-ratings. The object-level

program P that was used for the measurements shown in figure 6.5 is the standard logic programming benchmark of naive-reverse on a list of 30 elements, where naive reverse is defined as:

```
nrev([], []).
nrev([H|T], R) :- nrev(T, TRev), append(TRev, [H], R).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

This is, of course, the representation of P for I_0 and I_1. The version for I_2 uses the different representations for variables and conjunctions, but is otherwise the same. The LIPS-measurements reported in the first three columns of figure 6.5 are reliably reproducible within a margin of error of $\pm 10\%$, making the ratios in the other columns reliable within $\pm 20\%$. What is important in the table of figure 6.5 are not the absolute figures measured for I_0, I_1 and I_2 (the measurements for the different Prolog systems were taken on different machines), but rather the differences between the ratios I_0/I_1 and I_0/I_2 (i.e. I_1/I_2) for each Prolog system. This shows that I_1 runs an order of magnitude slower than the base-level Prolog system in which it is executed, while I_2 runs 4 orders of magnitude slower than its executing base-level.

We should explain here that the column for I_0 in figure 6.5 (and consequently the ratios involving I_0) are lower than the figures which are claimed by the vendors of these systems. This is because commercial Prolog systems are heavily optimised for the `nrev`-benchmark (by incorporating special case instructions in their virtual machine instruction set), resulting in an unrealistically high value for $\mathrm{LI}(I_0, nrev(n))$, which is not reached by these systems while executing other programs, such as I_1 and I_2. We therefore measured $\mathrm{LIPS}(I_0, nrev(30))$ not on the code for `nrev` listed above, but on the following procedurally equivalent code:

```
nrev([], []).
nrev([H|T], R) :- nrev(T, TRev), append(TRev, [H], R).

append([], L, L) :- !.
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3), !.
append(_, fail, fail).
```

This change eliminates most of the special case optimisations, resulting in a value for $\mathrm{LIPS}(I_0, nrev(30))$ which is only 25% of the speed quoted by the vendors, but which is a better reflection of the speed of the system.

The conclusion of these simple measurements (i.e. comparing $\mathrm{LI}(I_i, P(n))$ ratings for fixed $P(n)$ for $i = 0, 1, 2$) is that the bilingual interpreter runs 4 orders of magnitude slower than the mono-lingual interpreter.

All the above measurements are for an object-level program P that takes an input of fixed length n. The measurements reported in figure 6.6 show the results of measuring $\mathrm{LIPS}(I_i, nrev(n))$ in the Quintus Prolog system for varying values of n. As in the previous figure, the measurements of figure 6.6 are reliably reproducible within $\pm 10\%$. The prime

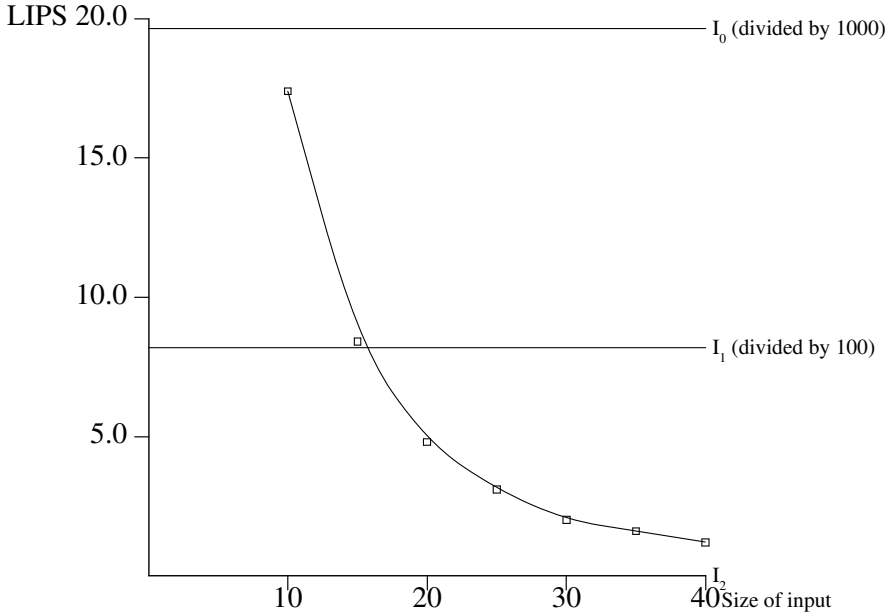


Figure 6.6: LIPS ratings on `nrev(n)`

interest of figure 6.6 is of course that both I_0 and I_1 show a constant LIPS-rating over varying input size, while I_2's LIPS-rating drops when the input increases. The reason for this different behaviour of I_2 is that both I_0 and I_1 can dereference their variables (Prolog variables in both cases) in constant time, whereas I_2's cost for dereferencing a variable grows with the size of the computation (since it handles substitutions explicitly, and they become larger for larger input).

The main conclusion from this second experiment measuring LIPS-ratings is that the inverse relation between the LIPS-rating of I_2 and the size of the computation contrasts sharply with the behaviour of I_1 and I_0, which does not depend on the size of the proof. This indicates that the loss of efficiency due to meta-level interpretation is much larger than is generally estimated in the logic-programming literature[2].

We will now turn to the second approach mentioned above, to measure the difference in overhead between I_1 and I_2, namely measuring the ratios $\mathrm{LI}(I_0, I_i(P(n)))/\mathrm{LI}(I_0, P(n))$ for $i = 1, 2$. The value of $\mathrm{LI}(I_0, P(n))$ can be easily computed for our standard benchmarks `append/3` and `nrev/2`. By looking at their code (given above), we find:

$$\begin{aligned} \mathrm{LI}(I_0, append(0))^3 &= 1 \\ \mathrm{LI}(I_0, append(n)) &= \mathrm{LI}(I_0, append(n-1)) + 1 \end{aligned}$$

or equivalently

$$\mathrm{LI}(I_0, append(n)) = n + 1 \tag{6.16}$$

[2]This cost is usually taken to be an order of magnitude, and constant in the size of the computation, as with the mono-lingual interpreter I_1 above. See for instance [Sterling and Beer, 1986].

[3]Of course *append/3* takes two input arguments, but only the first argument determines the complexity of the algorithm, thus justifying our notation *append(n)*.

by a simple induction argument on n. Similarly, for `nrev/2` we find

$$\begin{aligned} \mathrm{LI}(I_0, nrev(0)) &= 1 \\ \mathrm{LI}(I_0, nrev(n)) &= \mathrm{LI}(I_0, nrev(n-1)) + \mathrm{LI}(I_0, append(n-1)) + 1 \end{aligned}$$

or equivalently, again by induction on n, and using equality (6.16):

$$\mathrm{LI}(I_0, nrev(n)) = \tfrac{1}{2}(n+1)(n+2) \tag{6.17}$$

A similar counting argument can be done to find the values for $\mathrm{LI}(I_0, I_1(P(n)))$. Looking at the code for I_1 in figure 6.4a we see that I_0 needs 2 logical inferences for each or-branch in P interpreted by I_1 (the last clause in the code of I_1), 1 logical inference per and-branch (the second clause), and 3 logical inferences for each leaf in P's proof tree. Taking $P =$`append/3`, we get

$$\begin{aligned} \mathrm{LI}(I_0, I_1(append(0))) &= 3 \\ \mathrm{LI}(I_0, I_1(append(n))) &= \mathrm{LI}(I_0, I_1(append(n-1))) + 2 \end{aligned}$$

or, in closed form

$$\mathrm{LI}(I_0, I_1(append(n))) = 2n + 3 \tag{6.18}$$

and similarly, taking $P =$`nrev/2`

$$\begin{aligned} \mathrm{LI}(I_0, I_1(nrev(0))) &= 3 \\ \mathrm{LI}(I_0, I_1(nrev(n))) &= \mathrm{LI}(I_0, I_1(nrev(n-1))) + \mathrm{LI}(I_0, I_1(append(n-1))) + 2 + 1 \end{aligned}$$

or, in closed form:

$$\mathrm{LI}(I_0, I_1(nrev(n))) = n(n+1) + 4n + 3 \tag{6.19}$$

Having thus derived $\mathrm{LI}(I_0, I_i(P(n)))$ for $i = 0, 1$ and $P =$`append/3, nrev/2` and realising that $\mathrm{LI}(I_i, P(n))$ is the same for $i = 0, 1, 2$ we can now compute the ratio $\mathrm{LI}(I_1, P(n))/\mathrm{LI}(I_0, I_1(P(n)))$, which relates the speed of I_1 with the speed of I_0, as specified in equation (6.15). Combining (6.16) with (6.18) and (6.17) with (6.19) we get:

$$\begin{aligned} \frac{\mathrm{LI}(I_0, append(n))}{\mathrm{LI}(I_0, I_1(append(n)))} &= \frac{n+1}{2n+3} \\ \frac{\mathrm{LI}(I_1, nrev(n))}{\mathrm{LI}(I_0, I_1(nrev(n)))} &= \frac{\frac{1}{2}(n+1)(n+2)}{n(n+1)+4n+3} \end{aligned}$$

These ratios both approach $\frac{1}{2}$ for large n. This would indicate that I_1 is about twice as slow as I_0, whereas our previous measurements of LIPS ratings indicated that I_1 is an order of magnitude slower than I_0. This indicates a deficiency in the use of the logical inference as a unit of measurement, but even with this quantitative discrepancy, the above ratio is still qualitatively correct: it predicts correctly that the relation between the speed of I_1 and I_0 is constant, and independent of the size of the input n.

A similar counting exercise as above would, in principle, be possible in order to compute $\mathrm{LI}(I_0, I_2(P(n)))$, so that we could find a ratio between the speeds of I_0 and I_2, but the complexity of the code for I_2 (the code in figure 6.4b, plus the code for all the predicates

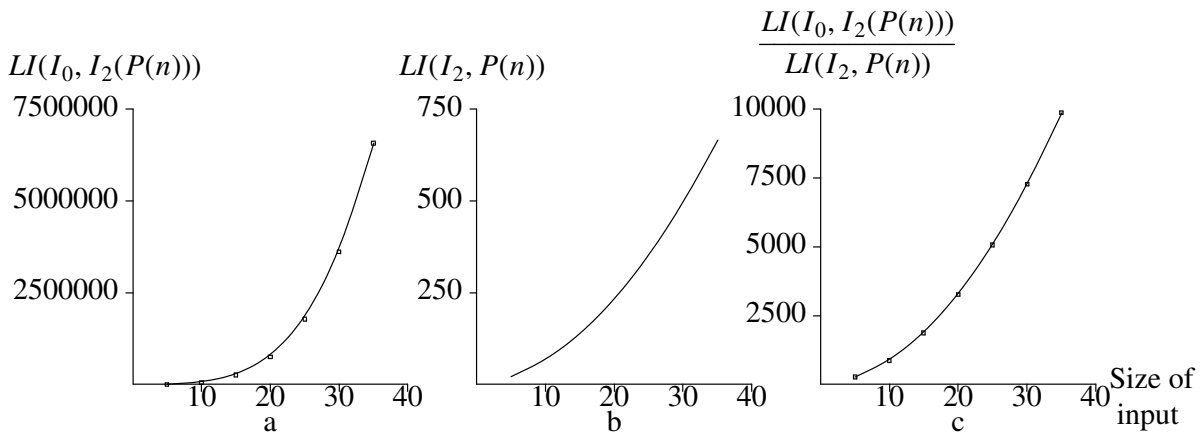


Figure 6.7: LI ratings and ratios for I_2 and I_0

mentioned there) makes this impractical. However, although impractical as a computation on paper, we can construct a program that executes I_2 and at the same time keeps track of the number of logical inferences made during this execution. This program produced the measurement of figure 6.7 which shows the values of both $\mathrm{LI}(I_0, I_2(P(n)))$ (figure a), $\mathrm{LI}(I_0, P(n))$ (figure b) and $\mathrm{LI}(I_0, I_2(P(n)))/\mathrm{LI}(I_0, P(n))$ (figure c) for $P =$`nrev` and for varying n. The third curve in this figure (figure c) shows the factor by which I_2 is slower than I_0 (in terms of logical inferences needed by I_0 to perform a given task). Again, as in figure 6.6, we see that the overhead of I_2 is not constant with the size of the computation, but rather grows with increasing input. The ratio in terms of LI-ratings of figure 6.7 is consistent with the LIPS-ratings of figure 6.6, showing an overhead of 4 orders of magnitude for input size $n = 30$.

Before we draw any conclusions from the above, we have to remark that these results are based on the most extreme version of a bilingual meta-level interpreter, namely with the full code of the meta-interpreter specified in the meta-level theory. An obvious optimisation would be to hardwire such predicates as `rename_vars/3, head/2, unify/3, body/2, instantiate/3` and `compose/3` (i.e. those predicates that are not part of the search strategy defined by the meta-level interpreter), rather than defining and executing them explicitly in the meta-level theory. Predicates that do affect the control strategy of the system, like `get_clause/2`, and `append/3`, and of course the definition of `solve/2` itself should be left explicit. It is difficult to say what exactly the overhead of I_2 would become with such a hardwired set of predicates, without doing a full and efficient implementation of them, which would be a serious task. The best approximation we can make using our experimentation environment is to regard these predicates as built-in predicates, i.e. to count each call to one of these predicates as 1 logical inference. This immediately reduces the value of $\mathrm{LI}(I_0, I_2(\mathit{append}(n)))$ to $12n + 11$ (by a similar counting argument as above), and thereby making the ratio $\mathrm{LI}(I_0, I_2(P(n)))/\mathrm{LI}(I_2, P(n))$ a constant one, although still an order of magnitude. However, this approximation is a simplistic one, since in general the execution costs of the hardwired predicates will not be constant, but will increase with the size of the proof, so a proper judgement on this issue has to be deferred to later work.

6.3 Implementation

Finally, we will briefly describe some aspects of the programs that were constructed to obtain all the measurements used in this chapter.

In order to obtain the LIPS-ratings (i.e. LI/*sec*) used in this chapter it was only necessary to measure the actual run-time of the programs, as explained above. This is because we are interested in the ratio

$$\frac{\text{LIPS}(I_1, P(n))}{\text{LIPS}(I_2, P(n))} = \frac{\text{LI}(I_1, P(n))/sec}{\text{LI}(I_2, P(n))/sec},$$

and since $\text{LI}(I_1, P(n)) = \text{LI}(I_2, P(n))$ we do not require the values of $\text{LI}(I_i, P(n))$, $i = 1, 2$, but only the respective run-times. These run-times can be measured using the Prolog evaluable predicate `runtime/2` which is built into most DEC-10 based Prolog systems such as the systems used in this chapter. However, in the systems used above, the implementation of this primitive relies on the Unix library facility *times* which suffers from a number of problems which causes the results to be not totally reliable:

- The time-resolution of the *times* library call is only 1/50 of a second, prohibiting the measurement of very small time intervals. For our purposes, 1/50 sec. is indeed a very coarse resolution: an 80KLIPS Prolog system such as Quintus can perform 1600 logical inferences in 1/50 sec.

- Due to the timesharing nature of the Unix system, the measurements of run-times will always be influenced by the use that other processes make of the machine's resources during the measurement. Furthermore, Unix is a virtual memory system. This implies that the run-time of a program is influenced by which virtual memory pages happen to be in our out of real memory at the time of the measurement, a factor not under the control of the programmer. All this is exacerbated by the fact that the particular machines used for our benchmarks were accessing their virtual memory (on disk) over a network which can influence the time needed to bring a virtual memory page in and out of real memory depending on the particular load on the network. Again, this factor is outside the control of the programmer. The `times` library routine can supposedly distinguish between the time spent on executing system instructions (for paging, and by other processes), and the time spent by the program itself. However, this distinction is not very precise in practice.

- A final problem with measuring run-times is the fact that some of the Prolog systems used in the benchmarks above (notably Quintus and BIMProlog) can decide to do garbage-collection at uncontrollable moments during the execution of a Prolog program. This means that the run-time for a particular user program can vary greatly from one measurement to the next, depending on whether or not the Prolog system happened to do a garbage collection during the execution of the program.

The result of these problems is that all measurements have to be taken over sufficiently large time intervals so that the coarse resolution will not matter, and the fluctuations due to system load, network load and Prolog garbage collections will "average out". Even then no greater accuracy can be obtained in LIPS-measurements than ±10%.

No such problems of accuracy exist with measuring the LI-rating of a particular program. This rating can be established through a mathematical induction argument, which closely follows the recursive structure of the program to be measured. However, very quickly programs become too complicated for such arguments to be carried out in practice. For this case we have constructed a program which automatically counts the LI-rating of an arbitrary predicate. In order to this, the program transforms any n-place predicate which is to be measured into an $n+2$-place predicate, with the extra two arguments, say i and j, representing two numbers such that after the predicate succeeds, the value $j-i$ will be the number of logical inference performed by the predicate. Calling the predicate with $i=0$ will give us j as the number of LI's performed by the predicate. For example, the definition of `nrev/2` used in the experiments above:

```
nrev([], []).
nrev([H|T], R) :- nrev(T, TRev), append(TRev, [H], R).

append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

would be transformed into:

```
nrev([], [], I, J) :- J is I+1.
nrev([H|T], R, I, J) :-
    nrev(T, F, I, I1),
    append(F, [H], R, I1, I2), J is I2+1.

append([], L, L, I, J) :- J is I+1.
append([H|L1], L2, [H|L3], I, J) :-
    append(L1, L2, L3, I, I1),
    J is I1+1.
```

This code transformation reflects exactly the definition of LI as given in section 6.2. Built-in predicates were counted as 1 logical inference, and treated specially by the transformation process. For example, the predicate `append_print/3`, a variation of `append/3` which also prints all elements of the first list:

```
append_print([], L, L).
append_print([H|L1], L2, [H|L3]) :- write(H), append_print(L1, L2, L3).
```

would be transformed into:

```
append_print([], L, L, I, J) :- J is I+1.
append_print([H|T1], T2, [H|T3], I, J) :-
    write(H),
    append_print(T1, T2, T3, I+1, I2),
    J is I2+1.
```

where the use of `I+1` instead of just `I` in the recursive call to `append_print` reflects the cost of the `write/1` predicate as 1 logical inference.

6.4 Related work

A number of other authors have done more or less systematic investigations into the overhead of meta-level interpreters. We will discuss three of these below.

O'Keefe in [O'Keefe, 1988] discusses a number of mono-lingual meta-level interpreters for Prolog, of which I_1 discussed above is an example. By using a different representation of the object-level theory he manages to increase the speed of a variation of I_1 as measured via LIPS($I_0, I_1(nrev(30))$) from 1 KLIPS (as measured above) to 5.5 KLIPS. This still leaves the overhead of his variation of I_1 well within the range of a factor 10–100 as concluded above. O'Keefe does not extend his investigations to either I_2 (the bilingual meta-level interpreter), or to measuring the value of LIPS($I_0, I_1(P(n))$) for some object-level program P as a function of increasing n.

All the experiments described above use interpreters $I_0, 1, 2$ with exactly the same control regime. As a result, we have measured only the overhead of each of the interpreters, without asking if this overhead could be off-set by possible savings made by writing meta-level interpreters that implement control regimes that are tailored to specific applications. Two related systematic studies ([Owen, 1988a] and [Lowe, 1988]) have been done in this area. Owen developed a series of meta-level interpreters for the interpretation of an object-level theory concerning protein topology, and Lowe used the same set of meta-level interpreters for the interpretation of an object-level theory concerning the simulation of electronic circuits. Using I_1 as above, Owen confirms our findings of an overhead of a factor 10–100 for LIPS($I_0, I_1(P)$)/LIPS(I_0, P). It is interesting that this figure still holds when interpreting more significant object-level programs P than just `nrev/2` or `append/3`. When Owen moves on to building special purpose versions of I_1 (i.e. a version of I_1 which do not mimic the behaviour of I_0), he eventually manages to improve performance by a factor of 2. His specialised version of I_1 ($I_{1,Lemma}$) is based on the use of object-level lemmas, and interprets the object-level theory twice as fast as I_0. This improvement in performance is rather small, and the reason for this is of course that $I_{1,Lemma}$ has to reduce the number of object-level logical inferences (LI($I_{1,Lemma}, P$)) significantly in order to make up for the enormous increase in cost of a single logical inference. This is illustrated by the fact that the improvement of the amount of object-level inference it does (i.e. LI(I_0, P)/LI(I_0, I_1P)) is more than a factor 300, although the run-time improvement of $I_{1,Lemma}$ is only a factor 2.

In [Lowe, 1988], the interpreters developed by Owen were applied to a different object-level theory. She also finds a value of 10–100 for LIPS($I_0, I_1(P)$)/LIPS(I_1, P), indicating that this factor is indeed stable across a large variety of object-level theories P. Out of a number of versions of I_1 developed by Owen, and applied to a number of different object-level theories (representing different electronic circuits), she finds only one meta-level interpreter (based on the dynamic ordering of goals ($I_1, Goals$)), and one object-level theory (representing an n-bit adder ($P_{adder}(n)$)) with an improved performance over $I_0(P(n))$. This improved performance is achieved because $I_1, Goals$ reduces the complexity of interpreting $P_{adder}(n)$ from exponential to linear, so that for large enough n the (linear) increase in cost of a single logical inference performed by $I_1, Goals$ is off-set by the exponential reduction in the number of logical inferences needed to interpret $P_{adder}(n)$.

The general conclusion from all this seems to be that the increase in costs per logical

inference is so high for even I_1 that this makes the practical application of meta-level interpreters for achieving increased efficiency through tailoring the control regime very difficult if not impossible. None of the work in the literature investigates bilingual meta-level interpreters like I_2, but, as shown in this chapter, it is to be expected that the behaviour of I_2 is both quantitatively and qualitatively worse than that of any variation of I_1.

6.5 Conclusions

In this chapter we have investigated the problem of the overhead of meta-level interpretation. In the first part of the chapter we have presented a model which shows that the amount of meta-level overhead can indeed offset the gains made by the reductions of the object-level search space, and in the second part of the chapter we have performed measurements which show that the actual size of the meta-level overhead is so large as to be a serious practical problem. Our experiments have shown that the overhead in bilingual meta-level systems is much higher than in mono-lingual systems, and that this overhead is much higher than is usually acknowledged in the literature.

Chapter 7

Partial evaluation

Although many of the papers in the literature dealing with the use of meta-level interpreters for control issues acknowledge the inefficiency that is inherent in the multiple layers of interpretation, very few of them offer any solutions to this problem. An important section of the limited work on reducing meta-level overhead in recent years has been based on the idea of specialising the general purpose formulation of the meta-level control regime with respect to the particular object-level theory that is being used in the system. Most of the work on this idea is based on the use of *partial evaluation* as an optimisation technique.

Work reported in, for example, [Venken, 1984], [Takeuchi and Furukawa, 1986], [Takewaki et al., 1985], [Safra and Shapiro, 1986], [Levi, 1988] and [Gallagher, 1986], is all based on this technique. In this chapter we will first describe this technique, and its application to meta-level interpreters in logic-based systems. In the second part of this chapter we will explore some of the limitations of this technique in the context of meta-level interpreters for logic-based systems, which remain largely undiscussed in the literature[1]. We will also discuss some heuristic improvements that can be made to partial evaluation to alleviate some of its limitations.

As a spin-off from the work on partial evaluation described in this chapter, a close relationship was discovered between partial evaluation and a machine-learning technique called *explanation-based generalisation* [Mitchell et al., 1986]. This work is published in [van Harmelen and Bundy, 1988].

7.1 A description of partial evaluation

For some considerable time, the functional programming community, and more recently the logic programming community, has been discussing a technique called *partial evaluation* as a program optimisation method (see [Futamura, 1971] for an early paper on partial evaluation, [Ershov, 1982] for partial evaluation in functional programming, and [Komorowski, 1982], [Venken, 1984], and [Takeuchi and Furukawa, 1986] for partial evaluation in logic programming).

[1]The main work described in this chapter was done in the first half of 1987. Since then a number of other authors in the literature have explored the limitations of partial evaluation, and have independently reached similar conclusions. Some recent work in the literature on the limitations of partial evaluation, in particular [Chan and Wallace, 1988] and [Owen, 1988b], will be discussed in the final part of this chapter.

The main goal of partial evaluation is to perform as much of the computation in a program as possible without depending on any of the input values of the program. The theoretical foundation for partial evaluation is Kleene's S-M-N theorem from recursive function theory [Kleene, 1952]. This theorem says that given any computable function f of n variables ($f = f(x_1, \ldots, x_n)$), and k ($k \leq n$) values $a_1, \ldots, a_k$ for $x_1, \ldots, x_k$, we can effectively compute a new function f' such that

$$f'(x_{k+1}, \ldots, x_n) = f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n)$$

The new function f' is a specialisation of f, and is easier to compute than f for those specific input values. A partial evaluation algorithm can be regarded as the implementation of this theorem, and is, in fact, slightly more general in the context of logic programming: it allows not only that a number of input variables are instantiated to constants, but also that these variables can be partially instantiated to terms that contain nested variables. Furthermore, a partial evaluation algorithm allows k in the above theorem (the number of instantiated input variables), to be 0, that is, no input to f is specified at all. Even in this case a partial evaluation algorithm is often able to produce a definition of f' which is equivalent to f but more efficient, since all the computations performed by f that are independent of the values of the input variables can be precomputed in f'. Thus, a partial evaluation algorithm takes as its input a function (program) definition, together with a partial specification of the input of the program, and produces a new version of the program that is specialised for the particular input values. The new version of the program may then be less general but more efficient than the original version.

A partial evaluation algorithm works by symbolically evaluating the input program while trying to (1) propagate constant values through the program code, (2) unfold procedure calls, and (3) branch out conditional parts of the code. If the language used to express the input program is logic, then the symbolic evaluation of the program becomes the construction of the proof tree corresponding to the execution of the program.

It is clear from this definition that partial evaluation is very close to normal evaluation, and, for the case of logic, therefore very close to theorem proving. What distinguishes partial evaluation from normal evaluation is the ability to specify only part of the input of a program. However, in the case of logic this is hardly a special case (after all, unbound variables are normal objects in a logic program), and this makes that partial evaluation of logic programs is very close to their normal evaluation. This close relationship between partial evaluation and normal evaluation is also clear from the definition of a very simple partial evaluator for Horn Clause logic given in [van Harmelen and Bundy, 1988]: the code given there for a toy partial evaluator is almost literally the same as the code for the interpreter for Prolog given in figure 6.4a of the previous chapter.

A special case of partial evaluation is when none of the values for the input variables $x_1, \ldots, x_k$ are given (in other words, $k = 0$). In this case, the partial evaluation algorithm cannot do as much optimisation of the input program, and as a result the new program will not be as efficient. However, the new program is no longer only a specialisation of the original program, but indeed equivalent to it. Thus, in this way partial evaluation can be used as a way of reformulating the input program in an equivalent but more efficient way.

As a simple example of partial evaluation, consider the following function in Lisp:

```
(defun assoc (key alist)
   (cond ((null alist) nil)
         ((eq key (caar alist)) (car alist))
         (t (assoc key (cdr alist)))))
```

This function accesses the standard Lisp assoc-list data-structure. If we specify a partial input, such as

```
alist = '((key1 . val1)(key2 . val2))
```

then we can partially evaluate `assoc`, using the following call to the partial evaluator[2]:

```
(peval '(assoc key '((key1 . val1)(key2 . val2))))
```

to return the derived program:

```
(defun assoc (key)
   (cond ((eq key 'key1) '(key1 . val1))
         (t (cond ((eq key 'key2) '(key2 . val2))
                  (t nil)))))
```

One problem with partial evaluation in general is that the partial evaluator has to handle uninstantiated variables. This is because the input of the source program is only partially specified and some of the variables in the source program will not have a value at partial evaluation time. In most programming languages it is hard to deal with uninstantiated variables, and the partial evaluator has to be very careful about what it does and does not evaluate.

This is exactly the reason why logic programming is especially suited for partial evaluation. Unification is a fundamental computational operation in logic programming, and handling uninstantiated variables in unification is no problem at all. In fact, uninstantiated variables arising from partially specified input[3]can be treated like any other term. For instance, in the example above, care had to be taken not to further evaluate the `eq`'s and `cond`'s, since the variable `key` was uninstantiated at partial evaluation time. However, in Prolog, the program `assoc`:

```
assoc(_, [], []).
assoc(Key, [[Key, Value]|_], Value).
assoc(Key, [_|Alist], Value) :-
assoc(Key, Alist, Value).
```

plus a call to the partial evaluator:

[2]This call to the partial evaluator only specifies half of its input: the partially specified input to the source program. The other half of the input to the partial evaluator (the actual definition of the source program) is assumed to be globally available in the execution environment of this call. This argument could be made explicit in the obvious way.

[3]In the context of logic programming G' is a partial specification of G if G' is subsumed by G. We also say that G' is an *instantiation* of G (notation: $G' \leq_{inst} G$) if there exists a substitution for variables in G, θ , such that θ applied to G gives G': $G' = \theta G$. G' is a *strict instantiation* of G (notation: $G' <_{inst} G$) if there is a non-empty substitution for variables in G, θ , such that $G' = \theta G$.

```
theory(t1,
        [o1(a) & h1(X) => c1(X, a),
         o2(a) & h2(X) => c1(X, a),
         o3(_, X) => h2(X),
         o3(b, c),
         o2(a)
        ]).
```

Figure 7.1: an object-level theory in Prolog

```
[1] proof(Goal, Theory) :-
         object_level_axiom(Goal, Theory).
[2] proof(Goal, Theory) :-
         object_level_inference(Goal, Theory, New_Goals),
         proof(New_Goals, Theory).
[3] proof([], _).
[4] proof([Goal|Goals], Theory) :-
         proof(Goal, Theory),
         proof(Goals, Theory).
```

Figure 7.2: a meta-level interpreter in Prolog

```
:- peval(assoc(Key, [[key1, val1], [key2, val2]], Val)).
```

partially evaluates into:

```
assoc(key1, [[key1, val1], [key2, val2]], val1).
assoc(key2, [[key1, val1], [key2, val2]], val2).
assoc(_, [[key1, val1], [key2, val2]], []).
```

without having to worry about evaluation at all, since even with an uninstantiated variable `Key`, the procedure evaluates to the equivalent specialised code, which now contains no further calls to be executed at run time.

This technique of specialising a program with respect to its (partial) input to derive a more efficient version, can also be applied in the special case when the source program is itself the definition of an interpreter (i.e. a meta-level program). This will then produce a version of the meta-level interpreter which is specialised for the particular object-level program that was given as input specification.

Example in Prolog: Consider an object-level theory as in figure 7.1[4], and a meta-level interpreter that specifies how to use these clauses, as in figure 7.2. This meta-level inter-

[4]The predicate names in the object-level theory of figure 7.1 are meant to suggest that the c_i are

```
[1]  proof(o1(a) & h1(X) => c1(X,a), t1).
[2]  proof(o2(a) & h2(X) => c1(X,a), t1).
[3]  proof(o3(X,Y) => h2(Y), t1).
[4]  proof(o3(b,c), t1).
[5]  proof(o2(a), t1).
[6]  proof(h2(c), t1).
[7]  proof(c1(c,a), t1).
[8]  proof(X & Y, t1) :-
          proof(X, t1),
          proof(Y, t1).
[9]  proof([X|Y], t1) :-
          proof(X, t1),
          proof(Y, t1).
[10] proof([], t1).
```

Figure 7.3: the programs after partial evaluation

preter assumes a Socrates-like architecture, as described in chapter 5, where the predicate `object_level_inference` generates all possible formulae derivable in one step from the input formula using the available object-level inference rules[5]. A full instantiation of the input to the meta-level interpreter consists of the object-level theory, plus the top goal that should be proved. So, if the above meta-level program is partially evaluated with the arguments:

```
Goal = c1(X, a)
Theory = t1
```

with the following call to the partial evaluator:

```
:- peval(proof(c1(X, a), t1)).
```

then, assuming that the inference rules Modus Ponens and Conjunction Introduction are in the object-level inference rules, the derived version of the meta-level program becomes:

```
proof(c1(c, a), t1).
```

conclusions to be derived by the system, the $\mathtt{h}_i$ are intermediate results to be derived by the system, and the $\mathtt{o}_i$ are observables to be obtained by the system from external sources (such as the user). These names are purely mnemonic to help present some of the arguments later on in this chapter, and do not have any further computational meaning.

[5]Actually, the meta-level interpreter from figure 7.2 does not quite follow the Socrates architecture. For the sake of simplicity this interpreter mixes object- and meta-level language, so that object-level substitutions do not have to be handled explicitly by the meta-level interpreter. However, this simplification does not effect any of the arguments in this chapter, and only serves to scale down the examples.

However, we don't want to specify the top goal of the query, since we cannot predict which goal we will want to prove. So, we underspecify the input to the source program (the meta-level interpreter): we leave the query uninstantiated, and only specify the object-level theory that we want to use. In the example above we would call the partial evaluator with

```
:- peval(proof(Goal, t1)).
```

giving us the remarkable transformed source program as shown in figure 7.3. This program contains the direct results for all the successful proofs that could be performed by the meta-level interpreter, namely:

- clauses [1]-[5] contain all the results derived via clause [1] of the meta-level interpreter (using `object_level_axiom`),
- clauses [6]-[7] contain all the results derived via application of Modus Ponens (via clause [2] of the meta-level interpreter),
- clause [8] contains a precomputed scheme for applying Conjunction Introduction (based on clauses [2]-[4] of the meta-interpreter),
- clauses [9] and [10] repeat the code from the meta-interpreter for iterating over conjunctive goals. This code will never be used by the partially evaluated version, since the iteration over conjunctive goals has already been precomputed in clause [8]. The fact that this superfluous code still appears in the partially evaluated version is due to the difference in status of clauses [1]-[2] and [3]-[4] of the meta-level interpreter: clauses [1]-[2] are meant to be called by the user of the meta-level interpreter, whereas clauses [3]-[4] are only meant to be called by the code itself. If this information had been conveyed to the partial evaluator (for instance by introducing a new predicate name), clauses [9]-[10] would not have occurred in the partially evaluated code.

The partially evaluated code from figure 7.3 is of course much more efficient than the original code from figures 7.1 and 7.2. Any of the facts mentioned in clauses [1]-[5] of figure 7.3 can now be proved in 1 logical inference[6] instead of taking 2 logical inferences[7]. Similarly, the facts from clauses [6]-[7] can now be proved in 1 logical inference, instead of 8 and 21 logical inferences respectively. The speedup for conjunctive goals is 3 logical inferences per conjunction. These speedups may be quite small as absolute figures, but taken as a proportion of the small amount of inference done by this toy example it amounts to an order of magnitude speedup.

7.2 Problems of partial evaluation

Although the above example indicates the power of partial evaluation, there are some serious problems associated with partial evaluation as a tool for reducing meta-level overhead.

[6]The measure of a logical inference is defined in chapter 6, section 6.2.

[7]Counting the predicates *object_level_inference/3* and *object_level_axiom/2* as built in predicates that can be executed in 1 logical inference.

These problems are not widely reported in the literature. In fact, most of the literature ignores these problems by using partial evaluation for the optimisation of only a very limited class of meta-level interpreters. Our experiments with partial evaluation indicate two main problems. The first of these is related to the definition of the object-level program (which is one of the arguments of the meta-level procedure that is instantiated at partial evaluation time), and the second problem is to do with the amount of information that is available to the partial evaluator.

7.2.1 Object-level programs that change at run time

In partial evaluation, programs are specialised with respect to their (partial) input. This gives a derived program that is specialised with respect to its input, and obviously this specialised program cannot be used to do computations on different inputs. In our specific case, the only part of the input to the source program (the meta-level interpreter) that is specified is the object-level theory. However, this object-level theory is likely to change while the meta-level interpreter is running. We are likely to want to add axioms to the object-level theory during the proof of a particular query, thereby invalidating the optimised version of the meta-level program.

An obvious solution to this problem is to follow [Reichgelt and van Harmelen, 1985, Reichgelt and van Harmelen, 1986] in distinguishing object-level knowledge from *object-level data*. The latter is object-level information that is dependent on a particular session (such as the values of the observable predicates $\mathtt{o}_i$ in the example theory `t1` above), and therefore will certainly change during a run of the system, whereas the former is object-level information that is not likely to change between sessions, such as the general rules linking the observable predicates $\mathtt{o}_i$ and the intermediate hypotheses $\mathtt{h}_i$ to the conclusions $\mathtt{c}_i$. We can now restrict our partial evaluation algorithm to evaluate only code that uses object-level knowledge. The execution of code that uses object-level data on the other hand has to be postponed until run-time.[8]

Furukawa and Takeuchi [Takeuchi and Furukawa, 1986] describe an alternative solution for the special case when the object-level theory grows monotonically. This involves constructing a version of a partial evaluator which is specialised for the meta-level interpreter plus the current version of the object-level theory, by applying the partial evaluator to itself with the meta-level interpreter and object-level theory as input. When a clause is added to the object-level theory, the specialised version of the partial evaluator can be used to construct both a partially evaluated version of the meta-level interpreter for the increased object-level theory, as well as a new version of the specialised partial evaluator, which is in turn to be used when the next clause is added to the object-level theory. Since this process can be performed incrementally, the overhead of repeated partial evaluation is greatly reduced. However, although this incremental approach might be useful during the development stages of a system, it is doubtful whether the price of repeatedly constructing

[8]This separation of the object-level level theories into statically and dynamically known subtheories is close to the multi-theory approach used in the Epsilon system [Coscai et al., 1988]. Although they do not use the terms "data" and "knowledge", their distinction of different object-level theories is very similar. A similar distinction is made in [Treur, 1988], for the purpose of proving formal properties about diagnostic expert systems. He uses the term *"symptoms"* where we use the more general term "data".

a new specialised version of both meta-level interpreter and partial evaluator at run time does not cost more than it gains, especially when the object-level theory changes frequently. In the context of the problem of changing object-level theories, Sterling and Beer [Sterling and Beer, 1986] talk about "open programs", which are programs whose definition is not complete, for instance because input data for the expert system needs to be provided at run time. They do not provide a solution for this problem, since they

> "assume that a goal which fails during partial evaluation time will also fail at run time, that is, we assume that a system to be partially evaluated is closed."

7.2.2 Lack of static information

An important distinction can be made between so called *static* and *dynamic* information. Static information is information which is part of, or can be derived from, the program code, whereas dynamic information is dependent on the run time environment of the program. For the purposes of partial evaluation we include the values of input variables supplied at partial evaluation time in our definition of static information. For example, in the following `or` statement

```
(or (eq x 'a)(eql y 2))
```

under the partial input specification

```
y = 3
```

both arguments to the call to `eql` are statically available (and therefore so is the result of the call to `eql`), whereas only one argument to the call to `eq` is statically available, since the value of `x` can only be dynamically determined.

[Beetz, 1987] distinguishes three different types of information that can influence the search strategy:

1. Information that is independent of the current problem and the current state of the problem solving process.
2. Information that is dependent on the current problem, but independent of the current state of the problem solving process.
3. Information that is dependent on both the current problem and the current state of the problem solving process.

Since both the second and the third type of control information will only be dynamically available, search strategies that use such information cannot be optimised by using partial evaluation[9]. Only search strategies that are independent of both the input problem and the current state of the problem solving process can make full use of partial evaluation. However, such search strategies are very weak and general, and do not typically play a very important role in expert systems applications. It is notable that all the examples given

[9] or only to a limited extent, and by using much more complex versions of partial evaluation, for instance by introducing suspended goals into the produced code.

in the literature on partial evaluation show programs that employ only the first type of information.

In order to analyse the problem of dynamic information in more detail, we can distinguish three techniques that are an integral part of any partial evaluator in order to produce more efficient code[10]:

1. branching out conditional parts of the code,

2. propagating data structures,

3. opening up intermediate procedure calls (unfolding).

We will argue that each of these techniques is crucially dependent on a large proportion of the information being statically available. If most of the information is only dynamically available, partial evaluation will generate quite poor results.

The first technique deals with *conditional branches in the code*. If the condition for such a branch cannot be evaluated at partial evaluation time, because its value is dependent on dynamic information, the partial evaluator either has to stop its evaluations at this point, or it has to generate code for both branches of the conditional, and leave it to the run time evaluation to determine which of these branches should be taken. Neither of these strategies is very successful: if a partial evaluator has to stop optimising the source code at the first dynamically determined conditional it encounters in the code, the resulting code may be not very different at all from the original code, and hence will not be any more efficient. The other strategy (generating code for all possible branches) is also usually not very attractive, given the high branching rates of most programs. This will result in very bulky output code (possibly exponential in the size of the original code), most of which will not be executed at run time. In the context of Prolog, this will mean a large number of clauses that will have to be tried at run time, even though most of them will fail.

The second technique (*propagating data structures*) tries to pass on data structures through the code in the program. This passing of data structures can be done both *forward* (for input values), and *backward* (for output values)[11]. Obviously, the forward passing of data structures only works for those parts of the data that have been provided statically as partial input. The backward passing of data structures is typically dependent on the values of the input, and is therefore also blocked if most information is only available dynamically. A special problem occurs with the so called built-in predicates that are provided by the Prolog interpreter. These predicates often depend on the full instantiation of a number of their arguments (e.g. `is`), and can therefore not be executed by the partial evaluator if the argument-values are not available. Other built-in predicates cause side effects that must

[10]Sometimes a fourth technique is included in this list, namely the so called "pushing down meta-arguments" (see for instance [Sterling and Beer, 1986]). This involves transforming a call to meta-level predicate such as `solve(object-level-pred(X), Subst)` into a call to the newly created object-level predicated `object-level-pred(X, Subst)`. However, this technique relies necessarily on the fact that the object-level and the meta-level languages are the same, and is therefore not included in this list.

[11]It is exactly this passing of data structures which makes logic programs so suited for partial evaluation, since both the forward and the backward passing is done automatically by the unification mechanism that is provided by the standard interpreter for the language.

occur at run time (e.g. `write`), and such predicates must also be suspended by the partial evaluator.

The third technique (*unfolding*) tries to insert code for procedure calls as 'in-line code', rather than explicitly calling the procedures at run time. This technique runs into trouble as soon as the source program contains recursive calls. Although a particular recursive program may in practice always terminate at run time, this is not necessarily the case at partial-evaluation time, due to the lack of static information. In the case of a logic program this means that the proof tree may contain infinite paths for some uninstantiated goals, and the partial evaluation would be non-terminating. Since these infinite computations only arise from the lack of information at partial-evaluation time, and do not occur at run time, we will use the phrase *pseudo-infinite* computation. Two different types of pseudo-infinite computation can be distinguished. The first type, *pseudo-infinitely deep* computation, is caused by programs whose recursive clauses always apply (at partial evaluation time), but whose base clauses never apply, due to the lack of static information. This gives rise to a proof tree with infinitely long branches. The second type, *pseudo-infinitely wide* computation, is caused by programs whose recursive clauses always apply, but whose base clauses also apply. This gives rise to a proof tree with infinitely many finite branches. A mixture of both types of pseudo-infinite computation is of course also possible. Pseudo-infinitely deep computation corresponds to a program that needs an infinite amount of time to compute its first output, and pseudo-infinitely wide computation corresponds to a program that computes an infinite number of answers (on backtracking, in the case of Prolog).

An example of both problems is the predicate `num_elem/2`, given below, which selects numeric elements from a list:

```
num_elem(X, [X|_]) :- number(X).
num_elem(X, [_|L]) :- num_elem(X, L).
```

If this predicate is partially evaluated with no input specified, then the base case will never apply, and an infinitely deep computation will result. If this predicate is partially evaluated with the first argument bound to a specific number, but the second argument still unbound, then the base case will apply, but so will the recursive clause, resulting in an infinitely wide computation[12].

In certain cases, the occurrence of infinitely wide computation does not need to lead to problems during partial evaluation, in particular, if it is known at partial evaluation time how many outputs are required of the source program. If it is known that at most n different outputs are needed from the source program, then the partial evaluator can unfold the proof tree of the source program until the base clauses have applied n times. A realistic example of this is where a predicate P is immediately followed by a cut in a Prolog program:

$$Q_1, \ldots, Q_i, P, !, Q_{i+1}, \ldots, Q_k$$

[12]Interestingly enough, if the predicate is executed with the second argument instantiated, then none of these problems occur, and the predicate can be fully computed at partial evaluation time. Notice that this example relies essentially on the use of the meta-logical predicate `number/2`. If a pure Prolog predicate is used, then lack of information can never result in the failure of a clause.

or more generally

$$Q_1, \ldots, Q_i, P, Q_{i+1}, \ldots, Q_j, !, Q_{j+1}, \ldots, Q_k$$

where all the conjuncts $Q_{i+1}, \ldots, Q_j$ are known to be deterministic[13]. In such a case $n = 1$, i.e. only 1 output will ever be required from P. This sort of analysis does of course presuppose that the partial evaluation algorithm has knowledge about properties of cut and of determinateness of predicates in the source program.

In the more general case, where such information about n is not known, or for pseudo-infinitely deep programs, it is necessary for a partial evaluation program to select a finite subtree from the infinite proof tree, in order to guarantee termination of the partial evaluation algorithm. Let π be a source program, θ an input substitution to π, $P(\pi, \theta)$ a partial evaluation procedure, and let $\pi(\theta) \downarrow$ mean that π terminates on input θ, then we would at least require P to terminate whenever π would terminate on θ (or on some instantiation of θ). Formally:

$$\forall\pi\forall\theta : (\exists\theta' \leq_{inst} \theta : \pi(\theta') \downarrow) \rightarrow P(\pi, \theta) \downarrow .$$

This can of course always be achieved by trivial means, such as not unfolding recursive predicates at all, or only unfolding them once (as in [Venken, 1984]), or in general only unfolding them to a fixed maximum depth. However, a more sophisticated solution would be to incorporate a stop criterion in the partial evaluation procedure that will tell us whether a branch of the proof tree for $\pi(\theta)$ is infinite. Thus, we need a stop criterion S such that:

$$\forall\pi\forall\theta : (\forall\theta' \leq_{inst} \theta : \pi(\theta') \uparrow) \leftrightarrow S(\pi, \theta). \tag{7.1}$$

As soon as S becomes true on a branch for $\pi(\theta)$, the partial evaluation procedure should stop. The problem with such a criterion S is that it amounts to solving the halting problem for Prolog, and that therefore it is undecidable. The halting problem for a given language L is to find a predicate H_L that will decide whether an arbitrary program P written in L will halt on an arbitrary input I or not:

$$\forall P\forall I : P(I) \uparrow \leftrightarrow H_L(P, I)^{14} \tag{7.2}$$

One of the fundamental theorems of the theory of computation states that this problem is undecidable for any sufficiently powerful language L. Prolog is certainly sufficiently powerful, since it is Turing complete [Tarnlund, 1977]. Since having S from (7.1) would also give us H_{Prolog} from (7.2), S must also be undecidable. This means that the best we can hope for regarding a stop criterion for partial evaluation is one that is either too strong or too weak. A stop criterion which is too strong will satisfy the $\leftarrow$ direction of (7.1), but there will be some π_0 and θ_0 such that

$$S(\pi_0, \theta_0) \wedge \exists\theta' \leq_{inst} \theta_0 : \pi(\theta') \downarrow,$$

in other words, S will tell us that π_0 will not terminate on θ_0 (or any instantiation of it), when in fact it would. This would result in stopping the unfolding of the partial evaluation

[13] A predicate is deterministic if, given an input, it computes exactly one output.

[14] The reader should be aware of a possible confusion: the stop criterion S from (7.1) is true not when $\pi(\theta)$ will stop, but when $\pi(\theta)$ will *not* stop, indicating that the partial evaluator *should be stopped*. By analogy, (7.2) has been formulated using $P(I) \uparrow$ instead of the usual $P(I) \downarrow$.

algorithm prematurely, thereby producing suboptimal results. Conversely, a stop criterion which is too weak will satisfy the $\rightarrow$ direction of (7.1), but there will be some π_0 and θ_0 such that

$$\neg S(\pi_0, \theta_0) \wedge \forall \theta' \leq_{inst} \theta_0 : \pi(\theta') \uparrow,$$

in other words, S will tell us that π_0 will terminate on θ_0 (or some instantiation of it), while in fact it would not. This would result in a non-terminating partial evaluation.

In the practical use of a stop criterion, a partial evaluator would keep a stack of goals that are unfolded during the expansion of the input program. If we call the original goal $G = G_0$, and we describe the stack of unfolded goals by $G_i(0 > i > j)$, then a number of useful stop criteria are:

1. unification: $\exists i < j \exists \theta : \theta G_j = \theta G_i$. We write $G_j =_{unif} G_i$.
2. instantiation: $\exists i < j \exists \theta : G_j = \theta G_i$, i.e. $G_j \leq_{inst} G_i$.
3. strict instantiation: $\exists i < j \exists \theta (\textit{non-empty}) : G_j = \theta G_i$, i.e $G_j <_{inst} G_i$.
4. alphabetic variancy: $G_j \leq_{inst} G_i$ and $G_i \leq_{inst} G_j$, i.e. $G_j =_{inst} G_i$ (G_i and G_j are identical up to renaming of variables).

It is not necessary to consider another variation, namely ($G_j >_{inst} G_i$) where G_i is a strict instantiation of G_j, since any chain of ever more general subgoals (a chain $G_0, \ldots, G_n$ where $G_i <_{inst} G_j$ if $i < j$) will always have a most general goal as its limit, and therefore such a computation must always terminate. However, we can have two different variations of [1], namely unification without occurs-check ([1a]) and unification with occurs-check ([1b]). These stop criteria relate to each other as follows:

$$([3] \vee [4]) \leftrightarrow [2]$$
$$[2] \rightarrow [1b] \rightarrow [1a]$$

Simple examples can show that none of these criteria performs satisfactorily. In particular, none deals satisfactorily with pseudo-infinitely wide computations. Consider for example a predicate like:

```
p1(X, Y).
p1(X, Y) :- p1(s(X), Y).
```

which generates on backtracking all terms $\mathtt{s}^n(\mathtt{0})$ after the call

```
:- p1(0, Y).
```

None of the above stop criteria is able to prevent a partial evaluator from looping while trying to partially evaluate `p1` with the first argument instantiated. The point here is not that we would expect a great optimisation from the partial evaluation (it is not clear what such optimisation could possibly be), but rather that the presence of a predicate like `p1` in any code makes the partial evaluation non-terminating. An example that illustrates the difference between some of the stop criteria is partially evaluating the above predicate `p1` with no input specified. In this case stop criterion [3] is too weak, and lets the partial evaluator loop infinitely, while criteria [4] (and by implication [2], [1a] and [1b]) properly halt the partial evaluator and reproduce the original code. However, the roles of [3] and [4] swap over on the predicate:

```
p2(0).
p2(s(X)) :- p2(X).
```

which will succeed on any input of the form $\mathtt{s}^n(\mathtt{0})$. Partially evaluating `p2` with no input specified will be properly stopped by [3] (and by implication also by [2], [1a] and [1b]), but not by [4] which will loop forever.

For a more realistic example, we can turn to the example that was used in the previous section to describe the use of partial evaluation for meta-level interpreters, shown in figures 7.1, 7.2 and 7.3. The partially evaluated code in figure 7.3 was computed from the code in figures 7.1 and 7.2 using stop criterion [3] (strict instantiation). Had we used the stronger stop criterion [2] (non-strict instantiation), the partial evaluator would have produced the same code in as figure 7.3, but with clauses [6]-[7] replaced by the following clause:

```
[6a] proof(X, t1) :-
          proof([Y => X, Y], t1).
```

This new clause represents a precomputed version of Modus Ponens, but the partial evaluator has stopped short of actually applying this rule, as in in figure 7.3, due to the stronger stop criterion. As a result, this new code is not as efficient as the code from figure 7.3. An advantage of the stronger stop criterion is that it takes significantly less time to execute the partial evaluator (the difference between the execution times of the partial evaluator with stop criteria [2] and [3] is more than a factor of 100).

A different situation occurred while computing the code for figure 7.5 from section 7.3.3. Fewer bindings for variables were known for that partial evaluation, since the object-level theory was not included in the input specification. As a result, the stronger stop criterion [2] had to be used to produce the result in figure 7.5. Any weaker stop criterion would result in either a non-terminating partial evaluation, or in code with many spurious branches.

7.2.3 Summary of problems

Summarising, we can say that although in principle a powerful technique, partial evaluation is rather restricted in its use for optimising meta-level interpreters for two reasons.

- Firstly, if the object-level theory is going to be changed at run time, at least part of the object-level theory cannot be included as input to the partial evaluation algorithm, thereby reducing the optimisations achieved by partial evaluation.
- Secondly, if most of the information in a program is only dynamically available (i.e. at run time), partial evaluation suffers from the following disadvantages:
  - If the source code contains conditional expressions, then a partial evaluator will either have to stop the optimisation process at that point, or produce very bulky code.
  - Data structures cannot be propagated throughout the code.
  - If the source code contains recursive procedures, then, unless specific stop criteria are programmed for particular predicates, a partial evaluator will either produce suboptimal code, or termination of the partial evaluator is no longer guaranteed.

It has to be stressed that although the above discussion quotes examples in Prolog, the problems are not due to this choice, and are fundamental to the concept of partial evaluation.

7.3 Heuristic guidance to partial evaluation

Although the problems discussed above seriously limit the applicability of partial evaluation as a tool for reducing meta-level overhead, a number of heuristic solutions can be found to alleviate the problems to a certain extent. The heuristics discussed below are all based on the idea that we will try to build a specific partial evaluator for a particular application, rather than a general, application independent one. More precisely, we can maintain the general framework for a partial evaluator as discussed above, but identify specific places in the algorithm where a user can tune the algorithm to suit a particular application. In particular, in the following subsections we will point out a number of places in the partial evaluation algorithm where we can insert specific knowledge about the behaviour of components of the program to be evaluated. In our case these components will be meta-level predicates, i.e. predicates occurring in the meta-level interpreter.

7.3.1 The stop criterion

An obvious candidate for removing over-generality is the stop criterion which is used to determine when to stop unfolding recursive predicates. This criterion can never be correct in the general case (since such a criterion would solve the halting problem for Prolog programs, and hence for Turing machines). As a result, any general criterion is either going to be too weak (i.e. not halting on some infinite recursion), or too strong (i.e. halting too early on some finite recursion). However, if we know the intended meaning of particular parts of a program we can construct specialised termination criteria that are just right for these particular procedures. As example, consider the definition of `member/2`:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

We know that this predicate is guaranteed to terminate as long as the length of the second argument is decreasing. A good specialised stop criterion for this predicate would therefore be:

$$\exists i, j : i > j, member_i(X, L_i) \wedge member_j(X, L_j) \wedge \|L_i\| \geq \|L_j\|,$$

where $member_i$ represents the i-th call to `member`, and $\|L\|$ is the length of list L. This stop criterion would successfully unfold all calls to `member/2` where the second argument is instantiated, but will not loop on those calls where the second argument is uninstantiated (due to a lack of static information). This corresponds to the notion that `member/2` will be used to test membership of a given list, and not to generate all possible lists containing a certain element. This criterion will even work for partially instantiated second arguments (remember that partial instantiation can be caused by a lack of static information). A call to `member/2` like

```
:- member(X, [1,2|L]).
```

will properly partially evaluate to:

```
member(1, [1,2|L]).
member(2, [1,2|L]).
member(X, [1,2,X|_]).
member(X, [1,2,_|T]) :- member(X, T).
```

The partial evaluation has generated all possible results based on the available static information, while stopping short of looping on the uninstantiated part of the input.

The above halting criterion is based on the intended meaning and use of the predicate `member/2`, and cannot be generally used, since it would again be either too strong or too weak for certain predicates. Consider for instance the predicate `nlist/2` which generates a list of length n:

```
nlist(0, []).
nlist(s(X), [_|T]) :- nlist(X, T).
```

(We use terms $\texttt{s}^n\texttt{(0)}$ for representing the number n to avoid problems with the built-in arithmetical predicates. More about this below). This predicate should not be stopped when its second argument is increasing in length, as with `member/2`, but rather when its first argument is increasing in depth. Such metrics should be devised where possible for predicates used in a meta-level interpreter, and the stop criterion should be specialised for these cases.

7.3.2 Operational predicates

A second heuristic that we can inject in the partial evaluation algorithm is a special treatment for certain predicates which are known to be easy to compute at run time, but possibly hard or impossible to compute with only static information. This idea is based on the notion of an *operational predicate* as introduced in the explanation-based generalisation algorithm [Mitchell et al., 1986] which turns out to be closely related to the partial evaluation algorithm (see [van Harmelen and Bundy, 1988]).

The partial evaluation algorithm should stop when encountering such an operational predicate (which is declared as such beforehand), no matter what amount of precomputation could potentially be done using the definition of such a predicate. For instance, it is possible that the definition of such a predicate has a very high branching rate, leading to an explosion of the size of the code generated by partial evaluation, while only one of the many branches would be chosen and computed at low cost at run time, pruning all the other branches. In such a case it is better not to generate the highly branched search space explicitly at partial evaluation time, but to leave it for run time computation.

An example of this specialised treatment of the partial evaluation algorithm for certain predicates is the standard logic programming technique where a predicate, when applied to a list, unpacks the list into its elements, and then takes a specific action for each of the elements in the list. If these specific actions have a very high branching rate, a good strategy for the partial evaluator is to precompute the process of unfolding the list into its elements

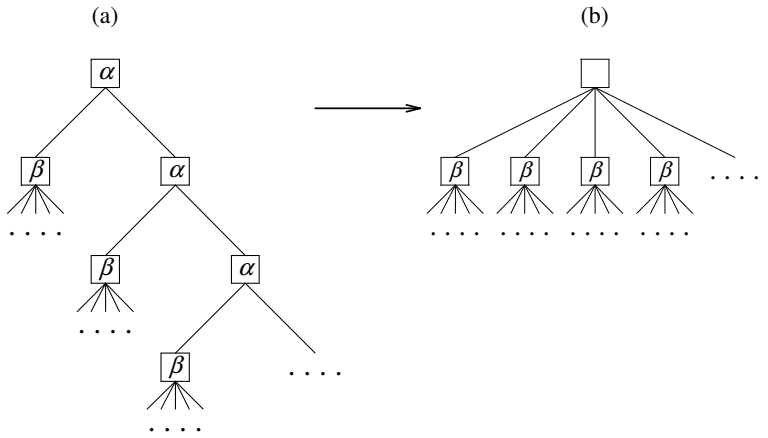


Figure 7.4: heuristically limited partial evaluation

```
proof(X, t1) :- object_level_axiom(X, t1).
proof(X, t1) :- proof([Y=>X,Y], t1).
proof(X&Y, t1) :- proof([X,Y], t1).
proof([X|Y], t1) :-
        proof(X, t1),
        proof(Y, t1).
proof([], t1).
```

Figure 7.5: the programs after restricted partial evaluation

(a deterministic operation), but to stop short of partially evaluating the actions taken for the individual elements. These actions will have to be performed at run time, when extra available dynamic information will possibly cut down the branching rate. Graphically, this optimisation process can be depicted as in figure 7.4. Figure 7.4a shows the search tree of original code before partial evaluation, with the α nodes doing the unfolding of the list, and the β nodes doing the highly non-deterministic actions for the individual nodes. If we designate β to be an operational predicate, the limited partial evaluation described above will produce code that has a search space as in figure 7.4b, showing that the limited partial evaluation still optimises the search space, but does not get bogged down in the explosive parts of it.

7.3.3 Restricted partial evaluation

A third heuristic that can be used to solve some of the problems associated with partial evaluation tackles the problem of a changing object-level theory. In section 7.2.1 we discussed how run-time changes to the object-level theory make it impossible to include the object-level theory as part of the input specification of the meta-level theory at partial

evaluation time. Two solutions to this problem were discussed (the distinction between object-level data and object-level knowledge, and the use of incremental partial evaluation). Another solution to the problem of run time changes to the object-level theory, would be to not include the object-level theory in the specialisation process at all. However, would this leave any input for specialisation of the meta-level interpreter? Looking at the components of a logic-based meta-level architecture in figure 4.3 from chapter 4, we see that if we do not want to include the object-level theory in the specialisation process, we are left with the object-level rules of inference. In terms of the example meta-level interpreter from figure 7.2, this would mean that we do not supply the object-level theory from figure 7.1, but that we do supply the definition of the predicate `object_level_inference`. This restricted version of the partial evaluation process would not generate a very efficient program like the one in figure 7.3, but the code in figure 7.5. Although this code is not as optimal as the code in figure 7.3, (since a lot of computation is still to be done at run time), it is still more efficient than the original version from figure 7.2. The new program will not be invalidated by run time changes, since the object-level theory was not used in the specialisation process (the predicate `object_level_axiom` will still be executed at run time, and was not precomputed by the partial evaluator, as it was in figure 7.3). The rules of inference (which were used in the specialisation process) are unlikely to be subject to run time changes.

A second form of restricted partial evaluation can be obtained by including neither the object-level theory nor the object-level rules of inference as input to the meta-level theory, and only partially evaluating the meta-level theory itself. This can still provide us with an optimised version of the meta-level theory if parts of the theory were expressed as domain independent knowledge. In general, domain independent knowledge will provide a concise and general way of representing knowledge, but much inference will be needed to apply it in a particular case. Using partial evaluation, it is possible to transform general but expensive domain independent knowledge into cheaper domain specific knowledge. We repeat here an example of this as given in [Clancey, 1983b] with a slightly simplified syntax. The following domain independent meta-level rule:

```
mentions(Rule1, Condition1) &
mentions(Rule2, Condition2) &
likely_to_be_true(Condition1) &
unlikely_to_be_true(Condition2)
-> use_before(Rule1, Rule2).
```

together with the domain dependent meta-level knowledge

```
likely_to_be_true("o1(a)").
unlikely_to_be_true("o2(a)").
```

could be partially evaluated to the domain specific meta-level rule:

```
mentions(Rule1, "o1(a)") &
mentions(Rule2, "o2(a)") &
-> use_before(Rule1, Rule2).
```

Furthermore, if we supplied the meta-level information:

```
mentions(axiom1, "o1(a)").
mentions(axiom2, "o2(a)").
```

the whole rule could be even further optimised to

```
use_before(axiom1, axiom2).
```

7.3.4 Mixed computation

Another heuristic to optimise partial evaluation is *mixed computation*. This involves declaring certain meta-level predicates to be executable at partial evaluation time. If the partial evaluator comes across such a predicate during unfolding, it does not unfold that predicate using its ordinary unfolding strategy, but rather it calls the hardwired interpreter that would normally execute the meta-level code (i.e. in our examples the Prolog interpreter) to execute the particular predicate. The resulting variable bindings are then taken into account during the rest of the unfolding process, but the predicate itself can be removed from the code.

7.3.5 Evaluable predicates

A final heuristic to be embodied in the partial evaluator concerns the evaluable predicates, as typically built into a Prolog system. These predicates can be divided into three types, and for each of the types a different partial evaluation strategy should be used.

- The first type of evaluable predicates are those that perform *side-effects* (e.g. input-output). These predicates can never be performed at partial evaluation time, and must always be postponed until run time.

- The second type of predicates are those that can be partially evaluated if certain conditions hold. These conditions are specific for each particular predicate. For instance, the predicate `var/1` (which tests if its argument is a variable), can be partially evaluated (to `false`, pruning branches from the code) if its argument is not a variable. The reason for this is of course that if the argument is not a variable at partial evaluation time, it will never become a variable at run time, since variables only become more instantiated. On the other hand, if the predicate `var/1` succeeds at partial evaluation time, it must remain in the code, since its argument might or might not have become instantiated at run time. Another example is the predicate `==/2` (testing if its arguments are the same Prolog object). This predicate can be evaluated (to `true`, allowing it to be removed) if it succeeds at partial evaluation time. The argument here is that if it succeeds at partial evaluation time, it will also succeed at run time (since objects that are the same can never become different again), but when the arguments are different at partial evaluation time the predicate should remain in the code, since the objects might or might not have become the same at run time[15]. A third example of this category is the predicate `functor/3` (which computes functor and arity of a Prolog term). This predicate can be partially

evaluated if either the first or both the second and third arguments are (at least partially) instantiated. Further criteria could be provided for a number of other built-in Prolog predicates.

- The final type of evaluable predicates are those which can be either fully evaluated (if they are fully instantiated at partial evaluation time), or translated into a number of simplified constraints. A good example of this type of predicates are the arithmetic predicates. Obviously, a goal like `X is 5+4` can be fully computed at partial evaluation time, as well as goals like `9 is X+4` and `X>10, X<5` (although a somewhat more sophisticated algorithm is required). In general a goal like `X is Y` ⟨`op`⟩ `Z`, where ⟨`op`⟩ is any of the functions `+`, `-` and `*` can be fully computed at partial evaluation time if at least 2 out of the 3 arguments are instantiated. Some types of calls cannot be fully computed at partial evaluation time, but can be transformed into simplified conditions, for instance `X<10, X<11` can be reduced to `X<10`, and the integer division `4 is X/3` can be transformed into `X>11, X<15`.

7.4 Implementation

In this section we will briefly discuss a number of technical issues concerning the implementation of a partial evaluator that was used for experimentation and to produce the examples presented in this chapter.

Most partial evaluators described in the literature follow more or less the same algorithm, and appendix A lists the central code for this algorithm. This code is fairly small (only 39 lines of Prolog), so we will not discuss all the details of it. The main outline of the code is as follows: the top-level predicate of the code is `peval/2` which takes a goal as its first (input) argument, and returns a list of specialised clauses for that goal as its second (output) argument. Initialising the recursion stack (used for loop-checking) to the empty list, `peval/2` calls `peval/3`, which, except in three special cases (which will be discussed later), retrieves all relevant clauses from the Prolog database and calls `peval_clauses/3`. All `peval_clauses/3` does is catch goals without any defining clauses or call `peval_clauses1/3` otherwise. This predicate iterates over all the clauses listed in its first argument, and for each clause it iterates over all conjuncts in the body of the clause, calling `peval/3` on each conjunct and recursing on the result, after having made sure that each or-parallel clause is evaluated in its own binding environment using new variables. This recursion will terminate when (1) clauses are encountered with no defining clauses, or (2) clauses are encountered with no subgoals in their body, or (3) if a loop is detected (see below). For more technical details, the reader is referred to the comments in appendix A.

Three separate features of this code are worth mentioning since they reflect directly some of the topics discussed in this chapter, and they are not generally found in this form in partial evaluators published in the literature. The first of these is the way the stop criterion is embodied in the code. This criterion is not hard-wired into the code of the partial evaluator, but is instead isolated in the `loop(Goal, Stack)` predicate, called in

[15] This criterion for `==/2` is quite different from that used in the partial evaluator described in [Priedites and Mostow, 1987], where it is incorrectly treated the same as `=/2` (unification), which can always be performed at partial evaluation time.

the first clause of `peval/3`. This predicate recurses up the `Stack` of previously evaluated goals, and checks if `Goal` and an element of this stack together satisfy the stop-criterion. The definition of `loop/2` can be changed through a software-switch to reflect different stop criteria such as the ones discussed in section 7.2.2. The stop criteria mentioned there can all be efficiently implemented in Prolog as follows:

```
[1] unify(G1, G2) :- \+ \+ (G1=G2).
[2] instantiation(G1, G2) :- \+ \+ (ground(G1), G1=G2).
[3] strict_instantiation(G1, G2) :-
        instantiation(G1, G2), \+ instantiation(G1, G2).
[4] alphabetic_variant(G1, G2) :-
        instantiation(G1, G2), instantiation(G2, G1).
```

Notice the use of double negation to avoid variable bindings and the use of the predicate `ground/1` to bind all variables in a term.

The second feature concerns the use of mixed computation, as mentioned in section 7.3.4. Particular predicates can be declared `executable`. This will be noticed by the 3rd clause of `peval/3`, which will execute that predicate using the Prolog interpreter (rather than partially evaluating it). The effect of this execution will be communicated through variables shared with other predicates. This technique of mixed computation can significantly speed up the partial evaluation process. Obviously, only those predicates can be used for mixed computation whose behaviour at partial evaluation time will be the same as at run time. In general this means that all input to such a predicate must be available at partial evaluation time, and that the predicate must be side-effect free. Finally, notice that the predicate `executable/2` which returns the results of executing a goal at partial evaluation time must return *all* solutions to that predicate, so that the definition of `executable/2` reads:

```
executable(Goal, Results) :-
    findall(cl(Goal, [], []),
    call(Goal), Results).
```

and not just

```
executable(Goal, cl(Goal, [], [])) :- call(Goal).
```

which would just return the first result of `Goal`, instead of all results.

The third feature of the partial evaluator of appendix A is the way it deals with built-in (evaluable) predicates. This is done by the second clause of `peval/3`. The definition of the predicate `evaluable(Goal, Flag)` is intended to reflect the conditions under which a `Goal` that is a built-in predicate can be evaluated at partial evaluation time, as discussed in section 7.3.5. The system will bind the `Flag` argument to one of three different values:

- `success_evaluable` (the 3rd clause of `unfold/3`): the built-in predicate could be executed and succeeded. In this case the predicate is removed from the code (since any possible output values will have been communicated through shared variables).

- `fail_evaluable` (the 4th clause of `unfold/3`): the built-in predicate could be executed and failed. In this case the current branch of the partial evaluation process is deleted.

- `not_done_evaluable` (2nd clause of `unfold/3`): the built-in predicate could not be executed at partial evaluation time. In this case the predicate is copied into the resulting code for execution at run-time.

Taking as examples some of the built-in predicates discussed in section 7.3.5, some clauses for `evaluable/2` are:

```
evaluable(write(_), not_done_evaluable).
evaluable(var(V), fail_evaluable) :- \+ var(V).
evaluable(var(V), not_done_evaluable) :- var(V).
evaluable(X==Y, success_evaluable) :- X==Y.
evaluable(X==Y, not_done_evaluable) :- \+ X==Y.
evaluable(functor(T, F, N), success_evaluable) :-
        (\+ var(T) ; (atom(F), number(N))), !, functor(T, F, N).
evaluable(functor(_, _, _), not_done_evaluable).
```

7.5 Related work in the literature

Much related work has been done in the last few years on partial evaluation and its application to meta-programming (e.g. the collections [Bjorner et al., 1987] and [Abramson and Rogers, 1988]). Some of these papers analyse and discuss the limitations of partial evaluation in a similar way as we have done in this chapter, and we will discuss two of these papers in particular.

A well known problem with the use of negation in logic programming is that it is only sound when applied to fully instantiated goals. If applied to partially instantiated subgoals, the negation is said to "flounder" [Lloyd, 1984], and produces unsound results. This problem is particularly severe during partial evaluation. A partial evaluation algorithm cannot unfold a negated subgoal if it is not fully instantiated. However, due to the lack of dynamic information at partial evaluation time, many negated subgoals will not be fully instantiated, thus hampering the performance of partial evaluation. As a result, the partial evaluation algorithm given in [Lloyd and Shepherdson, 1987] is restricted to either evaluate negated subgoals completely (if they are fully instantiated), or not at all otherwise. To solve this problem, [Chan and Wallace, 1988] give two techniques for dealing with negated subgoals. Their solutions are based on two separate techniques, one for eliminating negation from a program, and the other for splitting the program up into smaller pieces, so that the partial evaluation algorithm can optimise larger parts of the source program.

A second paper that explores the limitations of partial evaluation is [Owen, 1988b]. Owen applied partial evaluation to a number of meta-interpreters that were developed for a particular application, and compared the results of this with hand-coded optimisations of the same set of interpreters. This careful analysis revealed many problems with the practical use of partial evaluation, some of which have also been discussed in this chapter:

- Partial evaluation should not always suspend built-in meta-logical predicates (see section 7.3).
- Partial evaluation causes significant fruitless branches in the object-level program (see section 7.2.2).
- Partial evaluation systems always suspend the execution of Prolog's cut, even when this is not necessary (see the example about infinitely wide computation in section 7.2.2).

In order to deal with these problems, Owen proposes a number of enhancements to the partial evaluation algorithm. The most significant of these is what he calls a *folding transformation*, which folds a sequence of conjuncts into a new, uniquely named procedure. This is the opposite of the unfolding operation described in section 7.1. The main goal of this extra operation is to control the branching rate of the code produced by partial evaluation. However, the introduction of this new operation makes a partial evaluation algorithm non-deterministic (the algorithm will have to choose between different possible operations at each step), whereas this was not the case before. This introduces a search component in the partial evaluation procedure that was not present without the folding operation. Further extensions that Owen proposes to the partial evaluation algorithm are the merging of clauses with identical heads, or with heads that only differ in positions containing local variables, and rules that allow the treatment of cuts under certain conditions at partial evaluation time. Unfortunately, even with these and many other special purpose extensions to his partial evaluation algorithm, Owen found the results of partial evaluation on his meta-level interpreters suboptimal, and it would require open ended theorem proving and consistency checking to achieve the same results as his hand-coded optimisations.

Chapter 8

Partial reflection

The previous chapter discussed partial evaluation as a technique for reducing the overhead incurred by a meta-level interpreter. It was based on the idea of specialising a general meta-interpreter with respect to a particular object-level theory to obtain a specialised version of the meta-interpreter. In terms of the classification of meta-level systems given in chapter 3, partial evaluation does not affect the structure of the meta-level system it tries to optimise. If the original (unoptimised) system is a meta-level inference system, then the optimised version of that system is still a meta-level inference system, consisting of a meta-level interpreter where the inference of the system takes place, with the object-level inference being simulated at the meta-level.

In this chapter we will discuss an approach to the reduction of meta-level overhead based on the idea of taking a meta-level architecture of one type in the classification of chapter 3 (a meta-level inference system), but actually implementing it in terms of another type of architecture (namely an object-level inference system)[1]. Such an implementation of a meta-level inference system as an object-level inference system can be equipped with a largely hardwired interpreter, and will as a consequence suffer much less from the problem of meta-level overhead.

Whereas the partial evaluation technique discussed in the previous chapter was based on a transformation process, namely transforming a general meta-level theory into a meta-level theory specialised for a particular object-level theory, no such transformation process will be necessary for the approach discussed in this chapter. A meta-level inference system is not *transformed into* an object-level inference system, but rather is *implemented as* an object-level inference system. Thus, rather than ascribing *one* particular architecture to a system (as we have done so far), we now have to distinguish two architectures for a system, namely the *conceptual architecture*, which determines the properties of the system (as discussed in chapter 3), and the *implementation architecture*, which determines how the conceptual architecture is realised in practice. Usually, these two architectures are equated, and a system with a conceptual meta-level inference architecture is also implemented as a meta-level inference architecture (the Socrates system described in chapter 5 is an example

[1]Remember that in chapter 3 we defined an object-level inference system as one where there is no separate interpreter for meta-level expressions, but only a hard-wired (ie. not explicit) interpreter for object-level expressions whose behaviour could be influenced in a number of predefined ways by meta-level statements.

of this). However, this chapter is based on the idea that the implementation architecture can be different from the conceptual architecture, and in particular that it is possible to build a system with a conceptual meta-level inference architecture, but implemented as an object-level inference architecture.

In chapter 3 we argued that a system with a meta-level inference architecture is to be preferred over a system with an object-level inference architecture. Obviously, we shall have to take care not to lose the advantages of a meta-level inference architecture when we implement it as an object-level inference architecture. In this chapter we will show how we can achieve this objective by constructing the implementation architecture around a fixed set of "programmable steps". A large part of this chapter will be devoted to the description of these programmable steps.

8.1 Using mixed-level inference systems for reflection

Before we describe in detail how we can implement a meta-level inference system in terms of an object-level inference system, we will briefly look at the third category of meta-level systems described in chapter 3, the so called mixed-level inference system. We will argue why mixed-level inference systems are not suited as an implementation architecture for meta-level inference systems. Mixed-level inference systems are characterised by the fact that the activity in the system switches between object-level and meta-level on the basis of particular criterion (and this criterion varies between the different subtypes of mixed-level inference system, see chapter 3). In the context of logic-based systems, this process of switching between object-level and meta-level can be formalised as a reflection step, as already mentioned in section 2.4.2. Such steps, which link the object-level and the meta-level are usually stated as follows:

$$\frac{Meta \vdash_M prove(\text{“}Object\text{”}, \text{“}P\text{”})}{Object \vdash_O P} \quad (1) \qquad\qquad \frac{Object \vdash_O P}{Meta \vdash_M prove(\text{“}Object\text{”}, \text{“}P\text{”})} \quad (2)$$

where *prove* is the definition of a meta-level interpreter in the meta-theory $Meta$, P is an object-level predicate to be proven in the object-level theory $Object$, "$Object$" and "P" are the meta-level names of $Object$ and P in $Meta$, and $\vdash_{O,M}$ are the derivability relations at object- and meta-level. For the purposes of reducing meta-level overhead, the second reflection principle (rule (2)), the so-called *downward reflection principle*[2] is of most interest. This inference rule says that, in order to prove the meta-level goal $prove(\text{“}Object\text{”}, \text{“}P\text{”})$ in the meta-level theory we can prove the object-level goal P in the object-level theory. This inference rule does indeed replace meta-level inference by object-level inference, and can thus be used as an optimising device which reduces the amount of meta-level inference. A meta-level interpreter that uses this reflection principle would contain a clause like:

```
reflectable(Theory, Goal),
```

[2]To call rule (2) *downward* reflection is motivated by the backward-chaining use of this rule: in order to prove the meta-level goal $Meta \vdash prove(\text{“}Object\text{”}, \text{“}P\text{”})$ we have to prove the object-level goal $Object \vdash P$. A forward chaining use of these rules would suggest to call rule (1) downward reflection instead.

```
reflect-down(Theory, Goal, Subst)
-> prove(Theory, Goal, Subst)
```

where `reflectable` would embody the criteria that decide if a goal is heuristically suitable for downward reflection. As far as logical criteria are concerned any goal can be reflected (rule (2) does not state any restrictions on "P"), but in practice we might want to reflect down only certain goals, for instance goals which are known to be "easy" in some sense, and therefore do not need much meta-level guidance for their proofs. The predicate `reflect-down` would translate `Goal` (the meta-level name of an object-level formula) into the corresponding object-level formula, then call the object-level interpreter for `Theory` on that formula, and return the result as the variable binding `Subst`.

A system embodying this downward reflection principle would correspond to a subtask-management system as defined in chapter 3. The meta-level interpreter can dispatch certain subgoals to the object-level interpreter using the downward reflection principle, and wait for the results to come back. Consequently, such a system would suffer from the same problems associated with subtask-management systems as discussed in chapter 3, namely the so called black box effect: the object-level proof started by a downward reflection is not inspectable and not interruptable by the meta-level interpreter. This means that if the object-level proof turns out to be harder than expected (as encoded in the `reflectable` predicate), the object-level interpreter might get lost in an unpleasant search space for the reflected goal, without any guidance from the meta-level interpreter. If we want to remedy this problem, we will have to allow the object-level interpreter to hand back control to the meta-level interpreter during its execution, in other words, we want to allow *upward reflection* (rule (1)), as well as downward reflection. There are two ways of accomplishing this, but, as argued below, both of these ways turn out to be unattractive.

The first way of allowing upward reflection is to incorporate instructions for upward reflection in the object-level theory, using a `reflect-up` predicate. Two problems are associated with this approach. Firstly, it would force us into adopting an amalgamated representation language (as discussed in sections 2.4.2 and 3.1.3). After all, `reflect-up` is a meta-level predicate, but has to be incorporated into the object-level theory. For example, if we have an object-level predicate `p(X, Z)` as follows

```
p1(X, Y) & p2(Y, Z) -> p(X, Z)
```

then, if we want to prove `p2(Y, Z)` by upward reflection, we have to write

```
p1(X, Y) & reflect-up("p2(X, Y)") -> p(X, Z)
```

Simply writing

```
p1(X, Y) & reflect-up(p2(X, Y)) -> p(X, Z)
```

would not do, since `p2` (an object-level predicate) would occur in the position of a function-symbol, whereas `"p2(X, Y)"` (the meta-level name of `p2(X, Y)`) is a constant (of the amalgamated language), and can occur as an argument of `reflect-up`. Apart from forcing us into an amalgamated language, with all its complications of self-reference, another disadvantage of this approach is that it only allows for upward reflection in statically defined places in the object-level theory, and not on the basis of dynamically computed conditions,

which would be much more desirable. An alternative approach to upward reflection is to program such conditions into the (hardwired)[3]object-level interpreter. This will not force us into an amalgamated language for object-level and meta-level theories, and it will allow for the possibility of dynamic conditions triggering the upward reflections, but it will necessarily restrict the set of conditions to the predefined set that is coded into the object-level interpreter.

Summarising then, we can say that a mixed-level system as an implementation architecture for a meta-level inference system will either suffer from the black box effect, or force us into an amalgamated representation language, or hardwire the criteria used for upward reflection. Given these problems, it seems more attractive to look at the possibilities of an object-level inference system as the implementation architecture.

8.2 Using object-level inference systems for reflection

In chapter 3 we explained that object-level inference systems performed inference at the object-level, using an implicit interpreter which cannot be changed by the user. The only places where the user can affect the behaviour of the interpreter are a fixed number of points in the computational cycle of the object-level interpreter, where certain decisions regarding the search strategy of the interpreter are made. We shall call these steps in the computational cycle that can be affected by the user the *programmable steps* of an object-level inference system. An example of such a system was the Prolog system by Gallaire and Lasserre, discussed in section 2.4.1, where the programmable steps consist of goal-selection (conjunctive choices) and clause-selection (disjunctive choices). By analogy to the reflection of mixed-level systems discussed above[4], we shall call the computation of a programmable step *partial reflection*. "Reflection" because it involves a switch from the object-level interpreter to the meta-level (and back again), and "partial" because we do not reflect the whole proof of an object-level formula, as in rules (1)-(2), but only a part of the computational cycle for the proof of that goal[5].

It is to be expected that object-level inference systems do not suffer as much from the problem of meta-level overhead as meta-level inference systems, since only part of the computational cycle is performed at the meta-level (namely the computation of the programmable steps), and the main loop of the interpreter is hardwired, and can therefore be implemented efficiently at a low level in the system's architecture. As a further optimisation of the performance of object-level inference systems, we can provide default definitions of each of the programmable steps which can also be implemented in an efficient manner. The user is then allowed to override these default definitions by giving explicit formulations of alternative behaviour for some of the programmable steps. Obviously, when the user overrides the default behaviour, the system becomes less efficient, since the description of the required behaviour must then be explicitly interpreted, rather than executed in a

[3]The object-level interpreter is by definition hardwired. After all, an explicitly represented object-level interpreter would be a meta-level interpreter.

[4]and as a slightly playful reference to "partial evaluation"

[5]This use of the term *partial reflection* should not be confused with another use of this term in the literature. Sometimes partial reflection is used to refer to a situation with partial provability predicates $prove_n$ which model the provability relation, but only for formulae of some complexity n: $prove_n(\text{“}f\text{”}) \vdash f$.

hardwired form. It is therefore desirable that the user can override certain programmable steps, while leaving other parts unchanged. This would allow for a gradual degradation of the systems performance. The more parts of the default strategy the user overrides, the slower the system becomes. As an example we can again look at the system from Gallaire and Lasserre, discussed in section 2.4.1. As mentioned above, the programmable steps of this system are clause-selection and goal-selection. A reasonable default definition for these steps would be Prolog's normal top-down, left-to-right strategy, which can all be hardwired, so that the default version of the system need not run much slower than a single level, fully hardwired Prolog interpreter. However, it is possible for the user to redefine either of the two programmable steps, constructing, for instance, a clause-ordering interpreter. Of course the system need not only have one default behaviour. We can have a library of hardwired default behaviours. The user can then make variations on each of these built-in default behaviours by changing one or more components.

In order not to lose the advantages of having a meta-level inference system, it is important that such a system still satisfies the definition of a meta-level system: is all the behaviour of the system still represented explicitly (i.e. inspectable and modifiable) in the meta-level theory? Of the two components of the meta-level theory of a partial reflection system (the programmable steps and the main computational loop), the programmable steps are certainly represented explicitly. Their definition is available for inspection as part of the meta-level theory, and can also be modified by the user, with corresponding effects on the behaviour of the system. The fact that when these definitions correspond to a default definition the system does not use the meta-level formulation but rather a low level implementation of them for execution does not affect their inspectability (because a high level formulation is still available) nor their modifiability (because the system will switch to using the high level formulation when it differs from the default version). The other component of the meta-level theory however, the main computational cycle which chains the programmable steps together, is available at the meta-level for inspection only, and cannot be modified by the user, and is thus not explicit according to our definitions. However, as we will argue in the next section, it is possible to formulate this computational cycle in such a way that it will never be necessary for a user to modify this part of the meta-level theory. In other words, the formulation of the main computational cycle we will give below is so general that (almost) all possible control regimes can be expressed by only modifying definitions of the programmable steps, without having to modify the main computational cycle. As a result, the only way in which the system violates the definition of a meta-level inference system (namely the non-modifiability of the main computational cycle) is a harmless one, and will not affect any of the properties of the architecture that were discussed in chapter 3.

Thus, the question that will decide whether object-level inference systems are suitable as an implementation architecture for meta-level inference systems is the following: is it possible to design an object-level inference system with a sufficiently general set of programmable steps, such that it is possible to model the behaviour of a large class of meta-level inference systems via the programmable steps of the object-level inference system only? In the following sections we will give a definition of an object-level inference system based on partial reflection that is indeed general enough to model a large class of meta-level inference systems.

8.3 The definition of the partial-reflection interpreter

We will define the partial-reflection interpreter (and more importantly, the set of programmable steps of the interpreter) in terms of the operation each step of the interpreter performs on the object-level search tree. Therefore, we will first define the data type of this search tree. Then we will define the programmable steps of the partial-reflection interpreter, and finally we will define the overall structure of the interpreter.

We first define the basic data-types:

$$\begin{aligned} \texttt{formula} &= \{\texttt{meta-level names of object-level formulae}\} \\ \texttt{substitution} &= \{\texttt{meta-level names of substitutions for object-level} \\ &\qquad \texttt{variables}\} \cup \{\texttt{nil}\} \\ \texttt{theory} &= \{\texttt{meta-level names of object-level theories}\} \end{aligned}$$

Each of these data types are normally regarded as subsets of the set of constants of the meta-language needed to define a meta-level interpreter, and are therefore quite reasonable ingredients for our definition of a data type that is to represent an object-level search tree.

We will use the standard OR-tree representation for proofs: each node in the tree represents a choice point in the proof, and contains a list of goals that must be proved in order to prove the top goal. Proofs generate bindings for variables as their results, so nodes must also contain a substitution-field to record these results. Given the basic data types mentioned above, the data-type `node` can be defined in the obvious way as:

$$\texttt{node} = \texttt{formula}\star \times \texttt{node}\star \times \texttt{substitution}$$

where $\texttt{T}\star$ is notation for a (possibly empty) list of elements of type `T`. The fields of a `node` represent respectively:

- the list of open goals at this node in the search tree
- the children of the node in the search tree
- the substitution for object-level variables computed as the result of proving the list of goals

It will be useful to distinguish the following subtypes of the `node` type:

$$\begin{aligned} \texttt{leaf} &= \{\texttt{n} \in \texttt{node} \mid \texttt{children(n)} = \emptyset\} \\ \texttt{terminalnode} &= \{\texttt{l} \in \texttt{leaf} \mid \texttt{goals(l)} = \emptyset\} \\ \texttt{successnode} &= \{\texttt{t} \in \texttt{terminalnode} \mid \texttt{substitution(t)} \neq \texttt{nil}\} \\ \texttt{failurenode} &= \{\texttt{t} \in \texttt{terminalnode} \mid \texttt{substitution(t)} = \texttt{nil}\} \end{aligned}$$

This gives us the hierarchy of types as shown in figure 8.1: `terminalnode`s are the final leaves of the tree indicating either success or failure (`successnode`s and `failurenode`s respectively). `Leave`s are nodes at the bottom of the tree, but not necessarily `terminalnode`s. They can also be `leave`s that can further be expanded. As the above definitions show, the

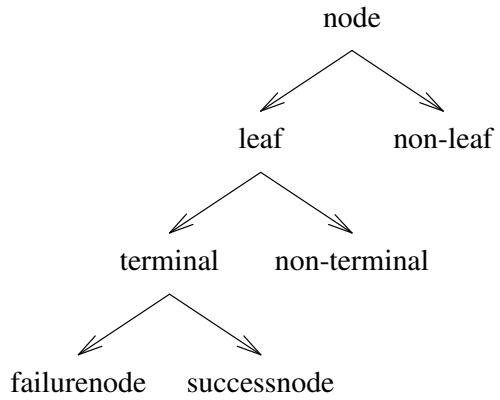


Figure 8.1: type hierarchy for search tree

```
prove(Goal, Theory, Tree, Status) :-
        initialise_tree(Goal, Tree),
        prove_loop(Tree, Theory, Status).
prove_loop(Tree, Theory, succeeded) :-
        succeeded(Tree).
prove_loop(Tree, _Theory, failed) :-
        failed(Tree).
prove_loop(Tree, Theory, Status) :-
        select_node(Tree, Theory, Leaf),
        select_goal(Leaf, Theory, Goal),
        expand_goal(Goal, Leaf, Theory, NewLeaves),
        filter_leaves(NewLeaves, Theory, FilteredLeaves),
        combine_leaves(FilteredLeaves, Leaf, Goal, FinalLeaves),
        hookup_leaves(FinalLeaves, Leaf),
        prove_loop(Tree, Theory, Status).
```

Figure 8.2: high level representation of the partial-reflection interpreter

`substitution` field of a node is used as an indicator of its status (failed or not), depending on whether its value is a non-nil substitution or not. Together with all the obvious constructor and destructor functions this defines the search tree as an abstract data-type.

We will now specify all the programmable steps of the partial-reflection interpreter that act upon this data structure. We specify each of the steps as a predicate, indicating the type of each argument of the predicate, and whether the arguments are input- or output-values (using the standard Prolog convention of '+' indicating input- and '–' indicating output-value):

```
succeeded(+Node)
```

```
failed(+Node)
select_node(+Node, +Theory, -NonTerminalNode)
select_goal(+NonTerminalNode, +Theory, -Formula)
filter_leaves(+Leaf*, +Theory, -Leaf*)
```

Although the behaviour of each of these programmable steps can be specified by the user (to override any possible default definitions), the semantics of each step is constrained to a certain extent, in order to allow all the separate predicates to be put together to form the partial-reflection interpreter. The constraints imposed on each of the programmable steps are described below. However, before we describe the specifications of the programmable steps, we give the full definition of the partial-reflection interpreter in which these steps play a role. This will make the description of the programmable steps somewhat easier. The interpreter is defined as in figure 8.2, and uses the following predicates apart from the programmable steps listed above:

```
prove(+Formula*, +Theory, -Node, -Status)
prove_loop(+Node, +Theory, -Status)
expand_goal(+Formula, +NonTerminalNode, +Theory, -Leaf*)
combine_leaves(+Leaf*, +NonTerminalNode, +Formula, -Leaf*)
hookup_leaves(+Leaf*, +NonTerminalNode)
initialise_tree(+Formula, -NonTerminalNode)
```

These predicates are part of the code of the partial-reflection interpreter that should be hardwired into the system for efficiency, and can therefore not be redefined by the user. It must be stressed that figure 8.2 is only written in Prolog notation for readability. For the architecture to be effective, this interpreter should be implemented at as low a level as possible, for maximum efficiency. Only the user specified redefinitions of the various components would be expressed in a logical language, and not the main computation loop itself. What is important however, is that the user can *think* of the interpreter as implemented as in figure 8.2. In terms of the definitions given in chapter 1, the code in figure 8.2 would be *inspectable*, but not *modifiable*.

We will now specify the behaviour of all the predicates that make up the interpreter of figure 8.2. The descriptions below marked with † are of predicates that are hardwired into the system, while the descriptions marked with • are the programmable steps mentioned above. These •-descriptions are therefore specifications of the behaviour of such a programmable step, and leave room for a more precise definition to be given by the user. For the hardwired predicates (marked with †) we will give the full specification.

† `Prove(Goal, Theory, Tree, Status)` is the top level predicate of the partial-reflection interpreter. After proving a `Goal` in an object-level `Theory`, the output argument (of type `node`) will return the search tree in the form of the top level node of the search tree from where all the other nodes are accessible. The fourth argument indicates whether the proof has succeeded or failed. We will discuss the success and failure behaviour of the interpreter in more detail below.

† `Initialise_tree(Goal, Tree)` takes a `Goal` as its input and produces a single `nonterminalnode` representing the root of the search tree.

† `Prove_loop(Tree, Theory, Status)` is the main loop of the partial-reflection interpreter, and distinguishes three cases: when the proof has succeeded, when it has failed or when the search tree is still open. It may seem as if the last (recursive) clause of `prove_loop` never terminates, since the recursive call is the same as the clause head. However, the underlying data structure for proof trees (as bound to the variable `Tree`), is an *incomplete data structure*, containing Prolog variables which become (partially) instantiated by procedures called in the body of `prove_loop`, thus changing the value of `Tree` between recursive calls. See [Sterling and Shapiro, 1986, chapter 15] for a further description of incomplete datastructures.

- `Succeeded(Tree)` and `failed(Tree)` decide when a proof has finished successfully or unsuccessfully. These predicates have no output values since they are used only as tests. A minimal specification of their behaviour is that `succeeded` can only succeed if at least one of the `leaves` is a `successnode`, while `failed` must succeed if all `leaves` of the tree are `failurenodes`. However, depending on the particular control regime, `succeed` does not have to succeed when one of the leaves is a `successnode`. In this way, the partial-reflection interpreter can search for multiple solutions, or for solutions that satisfy some extra-logical criterion. Similarly, `failed` can succeed before all nodes are `failurenodes`, thereby prematurely aborting the search process, and making the interpreter incomplete. More on this below when we discuss the combinatorial soundness and completeness of the interpreter.

- `Select_node(Tree, Theory, Leaf)` selects the next node in the search tree for continuation of the proof. Thus, `select_node` is a function from a search tree to a `leaf` of that search tree. It is this predicate that determines, for instance, whether the partial-reflection interpreter executes a depth first or a breadth first strategy. Furthermore, this predicate determines the backtrack behaviour of the interpreter. In other words, by choosing the next node in the search tree (which is an OR-tree) `select_node` affects the disjunctive choices made by the interpreter.

- `Select_goal(Leaf, Theory, Goal)` selects the `Goal` to be used in the next proof-step from the list of open goals in a node. Thus, `select_goal` is a function from `leaf` to a member of the list of open goals of that `leaf`. This predicate determines the order in which subgoals are solved. Thus, `select_goal` affects the conjunctive choices made by the interpreter.

† `Expand_goal(Goal, Leaves, Theory, NewLeaves)` takes the goal as selected by `select_goal` and applies all the inference rules of the object-level logic to it. This produces a list of new `leaves`, which can contain either `success-nodes` (if an inference rule established the truth of the goal, for instance because it is an axiom of the object-level theory), or `non-terminal` nodes for each inference rule that reduces the goal to a number of subgoals. If none of the inference rules apply, `expand-goal` returns a list containing just a single `failure-node`. Although the definition of this predicate is fixed, the user can of course affect the behaviour of this predicate indirectly by either changing the object-level theory or its inference rules.

- `Filter_leaves(NewLeaves, Theory, FilteredLeaves)` takes the output value of `expand_goal` (a list of `leaves`), and filters out unwanted expansions. Thus, the output value of `filter_leaves` is always a subset of its input value. This predicate can be used to implement various search heuristics such as avoiding unpromising branches in the proof etc.

† `Combine_leaves(FilteredLeaves, Leaf, Goal, FinalLeaves)` merges the new goal lists in the OR-nodes produced by `expand_goal` and `filter_leaves` with the open goal list of the selected node. For each of the new nodes a goal list is created which consists of the goal list of the selected node minus the selected goal plus the new goals found in the new node. This corresponds to the resolution step of a Prolog system. In this merging process the composition must be computed of the substitutions of the selected node and of each of the new nodes and the result must be stored in the substitution field of the corresponding output node. If the two substitutions contain conflicting variable bindings, the result of this process will be a `failurenode`, indicating the failure of that branch of the proof.

† `Hookup_leaves(FinalLeaves, Leaf)` joins the output list of `combine_leaves` with the existing tree (by updating the children field of the selected node). Notice that this action causes the selected node to be no longer a `leaf`.

An important feature of the partial-reflection interpreter described above is its behaviour on either success or failure of the object-level proof. Unlike the interpreters discussed in other chapters (such as the ones in figure 5.1, of section 5.1.2, and figure 7.2, of section 7.1) the partial-reflection interpreter does not model failure of the object-level proof as failure of the meta-level interpreter. Interpreters that do model object-level failure as meta-level failure are sometimes called *piggy-back* interpreters, and represent the widest class of interpreters published in the literature. Instead, the interpreter of figure 8.2 will always succeed, and report the success or failure of the object-level proof in the explicit `Status` argument. This makes the partial-reflection interpreter much more powerful than the ordinary piggy-back interpreters. It allows the user to control the backtracking behaviour of the object-level proof, rather than relying on the built-in backtracking behaviour of the meta-level interpreter. This treatment of object-level backtracking also means that the partial-reflection interpreter can be a completely deterministic program. Disjunctive choices at the meta-level can be programmed into the programmable step `select_node`, which chooses different nodes on different iterations of the interpreter, rather than choosing different nodes on backtracking, as would be usual in piggy-back interpreters. Any non-determinate behaviour of `select_node` is of a "don't care" nature ([Kowalski, 1979]). A consequence of this non-piggy-back way of handling object-level backtracking is that the partial-reflection interpreter will never fail, and similarly, that none of the predicates `select_node`, `select_goal`, `expand_goal`, `filter_leaves`, `combine_leaves` and `hookup_leaves` need ever fail. Of these, the predicates `expand_goal`, `combine_leaves` and `hookup_leaves` are hardwired, and can thus be ensured never to fail (as is implicit in their definition above). The non-failure of the user definable predicates `select_node`, `select_goal` and `filter_leaves` can easily be automatically checked by the system. One possibility is of course to allow failure of these user-definable predicates, and to perform

some hardwired default action when one of these predicates does indeed fail.

8.4 Adequacy of the partial-reflection interpreter

As already mentioned above, an architecture as described in the previous section will only be useful as an implementation architecture for a meta-level inference system if it is *adequate*, in other words, if a sufficiently large class of control regimes can be expressed through the formulation of the programmable steps. Ideally, we want every control regime expressible in a meta-level inference system to be expressable in the object-level inference system described above. Of course, the adequacy of a particular architecture can never be fully proven, because of the open nature of the phrase "every control regime expressable in a meta-level inference system". The best we can do is to argue convincingly for the adequacy of the partial-reflection interpreter.

In chapter 4, we argued that all the necessary components of a logic-based meta-level inference system could be summed up as the tuple $(O, I, \Sigma_g, \Sigma_d, \Sigma_t, T)$, with O the object-level theory, I the inference rules, $\Sigma_{g,d,t}$ the different heuristics, and the T the object-level search tree. All these components are indeed explicitly present in the partial-reflection interpreter: the object-level theory O is explicitly named by the `Theory`-argument of the interpreter, the inference rules I are embodied in the predicate `expand-goal`, the generative heuristics Σ_g correspond to the predicate `filter_leaves`, directional heuristics Σ_d to the predicates `select_node` and `select_goal`, termination heuristics Σ_t to the predicates `succeeded` and `failed`, and finally the object-level search tree T is explicitly named by the parameter `Tree`. Of course, the fact that all the essential components of a meta-level architecture are present in the partial-reflection interpreter does not in itself indicate that the interpreter is indeed adequate for expressing a wide range of control regimes. For this to be true, it is also necessary that a large number of versions of each of these components can be implemented in the interpreter. In principle, the full programming power of Prolog is available for the expression of these predicates, so this will not be a restriction. The only restriction on the programmability of the components (and in particular the components *SearchG* (`filter_leaves`) and *SearchD* (`select_node` and `select_goal`)) is their restricted input: they are defined as predicates over sets of leafs, and they can therefore only express local criteria, ie. criteria that are independent of such global properties of the proof such as the current depth, the total size of the proof tree, etc. This potential restriction could be removed by passing the proof tree as an additional argument into these predicates.

Summarising then, the facts that

- all the necessary components of a logic-based meta-level inference system are programmable,
- full Prolog is available for programming these components,
- the code for these components has (at least in principle) access to all information about the current proof and theory

indicate that the interpreter is indeed adequate for expressing a wide range of control regimes.

To illustrate the adequacy of the partial-reflection interpreter described above, we will give a few examples of how different control regimes can be programmed through the programmable steps of the interpreter.

A *depth-first*, *non-exhaustive* control regime for a Horn Clause interpreter can be formulated very easily as follows:

```
succeeded(Node) :- successnode(Node).
succeeded(Tree) :-
        non-leaf(Tree),
        children(Tree, Children),
        thereis(Child, Children, succeeded(Child)).

failed(Tree) :- failurenode(Tree).
failed(Tree) :-
        non-leaf(Tree),
        children(Tree, Children),
        forall(Child, Children, failed(Child)).

select-node(Node, Theory,Node) :-
        non-terminal(Node).
select-node(Node, Theory, Node) :-
        children(Node, Children),
        member(Child, Children),
        select-node(Child, Theory, Node).

select-goal(Node, Theory, Goal) :-
        goals(Node, Goals),
        member(Goal, Goals).

filter_leaves([], Theory, []).
filter_leaves([Node|Nodes], Theory, [Node|FilteredNodes]) :-
        goals(Node, Goals),
        (empty-node(Node);
         failure-node(Node);
         forall(G, Goals, (literal(G);object-level-axiom(Theory,G)))
        ),
        filter_leaves(Nodes, Theory, FilteredNodes).
filter_leaves([Node|Nodes], Theory, [Node|FilteredNodes]) :-
        filter_leaves(Nodes, Theory, FilteredNodes).
```

The criteria for `succeeded` and `failed` are simple: a tree counts as succeeded if it is a `successnode` or if one of its children contains a successnode. This represents the non-exhaustive nature of the control regime, i.e. the interpreter halts after finding one solution. Similarly, a tree counts as failed if it is a `failurenode` or if all of its children have failed.

The predicates `select-node` and `select-goal` represent the depth-first nature of the interpreter, selecting the first `non-terminalnode` found on a depth-first descent through the tree, and picking the first goal on the list of the selected node. The code for `filter-leaves` needs some explanation. For Horn Clause logic, the only two object-level inference rules we need are Conjunction Introduction and Modus Ponens. Because of that, we know that we can only prove goals that are either literals, or (if they are implications) are part of the object-level theory. This is a good example of how we can prune branches from the search tree which are generated by the object-level proof theory, but that we know to be unprovable, such as branches containing goals like `p->q->r` (which are unprovable from any Horn Clause theory).

The above control regime can of course be changed easily to a breadth-first interpreter for Horn Clause logic, by changing the definition of `select-node` to be

```
select-node(Node, Theory, Leaf) :-
        select-node([Node], Theory, Leaf).
select-node([Node|Nodes], Theory, Node) :-
        non-terminal(Node).
select-node([Node|Nodes], Theory, Leaf) :-
        children(Node, Children),
        append(Nodes, Children, NewNodes),
        select-node(NewNodes, Theory, Leaf).
```

in other words: we pick the first `non-terminalnode` found on a breadth-first descent through the tree.

Similarly, we can change the non-exhaustive nature of the control regime to an exhaustive one by changing the criteria for success and failure:

```
succeeded(Tree) :-
        one-success(Tree),
        all-terminal(Tree).
failed(Tree) :-
        \+ one-success(Tree),
        all-terminal(Tree).
```

where `one-success` descends the tree and succeeds when a `successnode` is found, and `all-terminal` descends the tree and succeeds if all nodes visited are `terminalnode`s.

In a similar vein we could implement other frequently used control strategies such as iterative deepening, best first search, confirmation or elimination strategies, branch and bound search, etc.

After having given these definitions for the programmable steps in Prolog, it is important to remember that, if the above definitions were default definitions, they would not be executed using the Prolog formulation but using some equivalent formulation in a more efficient (lower level) language instead. Only if the user would override the defaults by altering the above Prolog formulations would the system execute the explicit meta-level definitions.

One obvious optimisation can be applied to the partial reflection interpreter on the basis of these formulations of control regimes. As can be seen from the definitions of

`succeeded`, `failed` and `select_node`, these predicates often have to traverse the whole tree in order to arrive at the leaves of the tree where they do their real work. In a naive implementation this would be of exponential cost, thus defeating the purpose of the partial reflection interpreter (which is to reduce meta-level overhead). An obvious cure for this problem is to make available a built-in predicate which gives access to the current set of leaves of the search tree. This can be implemented without repeated traversal of the tree (for instance incremental updating of the set of leaves after tree expansion), and the user can then use this predicate in the formulation of control regimes. Taking for instance the depth-first control regime formulated above, and assuming that `leaves(L)` will unify `L` with the current leaves of the search tree, we can reformulate the control regimes as follows:

```
succeeded(Tree) :-
    leaves(Ls),
    member(L, Ls),
    successnode(L).
failed(Tree) :-
    leaves(Ls),
    forall(L, Ls, failurenode(L)).
select_node(Tree, Theory, L) :-
    leaves(Ls),
    member(L, Ls),
    non-terminal(L).
```

The definitions of `select-goal` and `filter-leaves` remain unchanged.

8.5 Combinatorial soundness and completeness

In chapters 3 and 4 we defined the notions of combinatorial soundness and completeness of a meta-level system. Combinatorial soundness and completeness referred to whether a meta-level interpreter produced a subset or a superset of the logical inferences derivable from the object-level theory. As noted in chapter 5, the meta-level of a system like Socrates has no facilities for guaranteeing either the combinatorial completeness or soundness of an interpreter. Because the form of the interpreter in an object-level inference system as described above is much more constrained than in a meta-level inference system like Socrates (after all, an interpreter in Socrates could be expressed using the full power of Horn Clause logic, whereas the basic structure of the partial-reflection interpreter is fixed, as in figure 8.2), it becomes possible to include automatic checks on the combinatorial completeness and soundness of the partial-reflection interpreter.

It is obvious from the examples in the previous section that the partial-reflection interpreter is by no means guaranteed to be complete. Any of the predicates `select_node, select_goal, expand_goal` and `filter_leaves` can be programmed not to select some nodes or not to produce all expansions, or to throw away some of the logically valid inferences. This is of course a desirable feature of the interpreter. In this way we can implement heuristics that cut out large parts of the search space. This might involve cutting out parts

of the search space that contain solutions that are too hard to find, or solutions that do not satisfy certain criteria. However, it is possible to say something about the conditions under which the partial-reflection interpreter will be complete. This completeness obviously depends on the behaviour of the programmable components of it. The interpreter is complete (with respect to the set of object-level inference rules I) if all of the following conditions are satisfied:

- `succeeded` succeeds only if all `leaves` of the tree are `terminalnodes` (i.e. the interpreter does not terminate before all branches are explored).
- `failed` succeeds only if all `leaves` of the tree are `failurenodes` (i.e. the interpreter does not terminate before all branches are explored).
- `filter_leaves` returns the complete set of input nodes (i.e. no branches are deleted from the search tree).

The situation with combinatorial soundness is different. Unlike completeness, soundness is a property we might want to be able to enforce, and it would be useful if we could guarantee the partial-reflection interpreter to be sound. Apart from the restrictions imposed by the typing of the predicates and those described in the definitions of the predicates given above, the following restrictions ensure the soundness of the interpreter:

- `succeeded` succeeds only if at least one of the `leaves` is a `successnode`.
- `failed` succeeds if all `leaves` of the tree are `failurenodes`.
- `filter_leaves` returns only nodes that are in the set of input nodes (i.e. no branches are added to the search tree that are not provable using the object-level inference rules)

Combining these sets of conditions, we get the conditions under which the interpreter is both combinatorially sound and complete:

- `succeeded` succeeds only if all `leaves` of the tree are `terminalnodes` and at least one of the `leaves` is a `successnode`.
- `failed` succeeds if and only if all `leaves` of the tree are `failurenodes`.
- `filter_leaves` returns exactly the set of input nodes (i.e. `filter_leaves` is the identity operation).

These criteria are such that it would be very simple to automatically check any of the above three sets of constraints, thereby ensuring any combination of combinatorial soundness and completeness of the interpreter.

8.6 Efficiency of the partial-reflection interpreter

Since the main motivation for the partial reflection interpreter is a reduced overhead for meta-level interpretation, we should compare its behaviour with the fully explicit bilingual meta-level interpreter as measured in chapter 6. There we measured both LIPS$(I_2, P(n))$ and LI$(I_0, I_2P(n))$, i.e. the number of logical inferences per second performed by the bilingual meta-level interpreter I_2 on some object-level program P with input of length n, and the number of logical inferences performed by the base-level Prolog system in order to execute I_2 interpreting $P(n)$ (assuming that I_2 was expressed in Prolog and could therefore be interpreted by I_0).

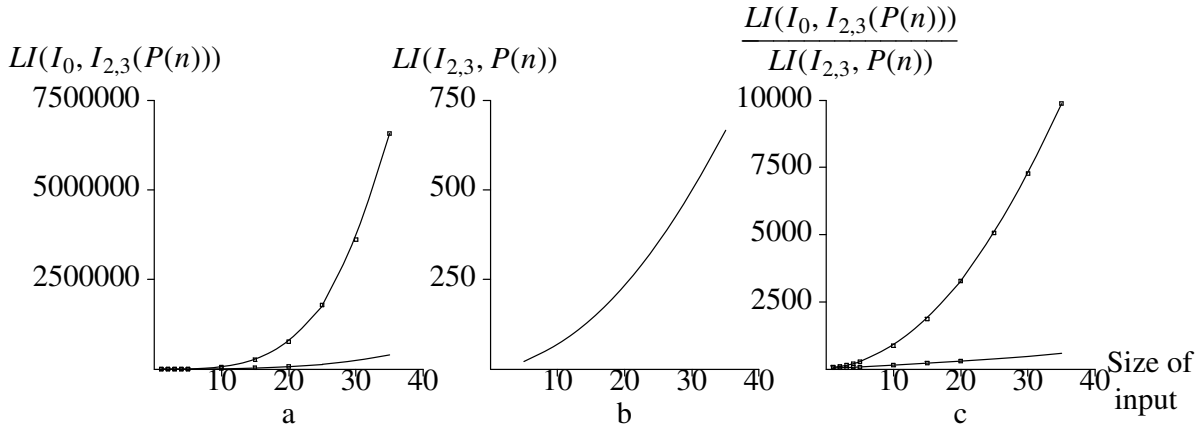


Figure 8.3: LI ratings and ratios for $I_{2,3}$ and I_0

If we denote the partial-reflection interpreter described above by I_3, we should ideally measure LIPS$(I_i, P(n))$ and LI$(I_0, I_iP(n))$ for $i = 2, 3$, and the ratios between these numbers in order to get an idea of the reduction in meta-level overhead achieved by using I_3 instead of I_2. However, a full and efficient implementation of I_3 is a significant task, including the encoding of an efficient unification algorithm for the object-level language and an efficient representation of the object-level theory. In order to avoid such a major implementation effort, we have implemented I_3 using the definition in Prolog given above in figure 8.2. Of course we would not want to calculate the logical inferences made by predicates which are supposed to be hardwired (such as `expand_goal`, `combine_leaves` and `hookup_leaves`), and in order to avoid this we can count each call to such predicates as a single logical inference (equating their cost with any built-in predicate). Of course, this simulation oversimplifies the cost of the hardwired predicates in the partial reflection interpreter to a single logical inference, and it also make it impossible to mention LIPS rates, since the predicates are not actually hardwired, but coded explicitly in Prolog, thus taking up more time. As a result, we can only measure LI$(I_0, I_i(P(n)))$ for $i = 2, 3$. For the purpose of these measurements the partial reflection interpreter described above was implemented using the code from figure 8.2, plus additional code to implement the hardwired predicates `expand_goal`, `combine_leaves` and `hookup_leaves`. For the predicate `expand_goal` this required firstly a representation of the object-level theory, which is represented using the

ground Prolog clauses. These Prolog clauses represent the meta-level names of object-level clauses, rather than the object-level clauses themselves. For instance, the object-level sentence $\forall$`x p(x)` $\rightarrow$ `q(x)` in object-level theory `t1` would be represented as:

```
object_theory(t1, p(var(x))->q(var(x))).
```

We then need a unification algorithm for formulae in this representation (i.e. using `var(x)` as a variable), returning explicit substitutions for these variables, (as needed to produce the substitution field of nodes in the search tree), and routines to manipulate these explicit substitutions (such as composition and application of substitutions). A second prerequisite for the predicate `expand_goal` is the representation of the object-level rules of inference, which are represented as Prolog clauses of the form

```
infers(Antecedent, Consequent)
```

with Prolog variables taking the role of meta-logical variables, and lists being used to represent the meta-logical connective '`,`'. For example, Modus Ponens would be represented as

```
infers([P, P->Q], Q).
```

The results of measuring LI$(I_0, I_3(P(n)))$ using the above implementation for I_3 are shown in figure 8.3. This figure repeats the results for I_2 from figure 6.7, and adds the results for I_3, using $P = nrev$. Notice that figure 8.3b shows only one line since LI$(I_i, P(n))$ has the same value for $i = 2, 3$ since I_2 and I_3 execute the same control regime. Figure a shows that the simulated version of I_3 runs an order of magnitude faster than I_2. More importantly, figure c shows that the overhead of I_3 does not increase so rapidly with the size of the input, but is instead a smaller and more constant factor.

8.7 Comparison with Eshghi's interpreter and conclusions

The meta-level interpreter described in [Eshghi, 1986] is very similar in nature to the partial-reflection interpreter described above, and in fact served as part of the inspiration for this chapter. However, there a number of aspects in which the interpreter described here is more general than the one in [Eshghi, 1986].

Our partial-reflection interpreter is more general in the sense that it allows for a richer object-level language than just Horn Clause logic. In fact, it is completely independent from the logical language used at the object-level (using the same relation between object-level and meta-level language as in Socrates). As a result, both the object-level language and the set of inference rules for this language can be changed independently of the control strategy expressed in the partial-reflection interpreter.

In our partial-reflection interpreter the criteria for success or failure are definable by the user, unlike the hardwired criteria of Eshghi's interpreter. This allows for extra flexibility in the formulation of control strategies. Eshghi argues (correctly) that his interpreter is an extension of the work of [Gallaire and Lasserre, 1982], because their interpreter only

allows the user to define the clause-selection and goal-selection strategies of the interpreter, whereas his interpreter also gives the user control of the backtracking strategy. Similarly, our interpreter is an extension of Eshghi's, since it includes the termination criteria in the user definable predicates. In terms of the representation of a meta-level inference system as a tuple $(O, I, \Sigma_g, \Sigma_d, \Sigma_t, T)$, Eshghi has no programmable step corresponding to Σ_t. programmable steps

Eshghi seems to argue for a system that enforces both combinatorial completeness and soundness. We disagree with this claim as far as completeness is concerned. As argued above, an essential feature of practical control regimes is often their *in*completeness.

Finally, although Eshghi's interpreter is much like the one presented above, he gives no motivation for the set of programmable steps that his interpreter offers, whereas we have argued that our set of programmable steps is motivated by the analysis of the general architecture of meta-level inference systems from chapter 4. As a result, we can have some confidence in the adequacy of our interpreter, whereas Eshghi gives no such argument.

In this chapter we have formulated an object-level inference system with a set of programmable steps (i.e. programmable by the user in the meta-level language of the system), that is powerful enough to model a large number of meta-level interpreters (as formulated in a meta-level inference system), but that can be implemented much more efficiently than a meta-level inference system through the technique of partial reflection (i.e. we only execute the programmable steps at the meta-level and provide a hardwired definition for the main loop of the interpreter). The performance of this system can be made even better by providing hardwired default definitions for each of the programmable steps, which can possibly be overwritten by the user.

So far, we have not compared the partial reflection interpreter described in this chapter with the partial evaluation technique discussed in the previous chapter. Does the partial reflection technique not suffer from the same problems as partial evaluation? The problems with partial evaluation all stemmed from the distinction between partial evaluation time and run-time. The object-level theory might be subject to run-time changes not known at partial evaluation time, and a number of problems arose from the lack of input known statically at partial evaluation time instead of dynamically at run-time. In the context of a partial reflection system, no such distinction between static and dynamic information exists. There is no pre-runtime transformation stage, and as a result none of the problems associated with partial evaluation arise with the partial reflection architecture.

Apart from an implementation architecture for meta-level inference systems, the partial-reflection interpreter defined above has advantages which are quite independent from any efficiency considerations:

- It does not treat object-level failure as meta-level failure, and therefore allows explicit control of the backtracking strategy of the object-level proof, instead of relying on the hardwired backtracking behaviour of the meta-level interpreter.

- It offers the possibility for automatically ensuring combinatorial soundness and completeness of the interpreter, whereas this task is very hard (if not impossible) in a meta-level inference system where the formulation of a meta-level interpreter can be an arbitrarily complex program in a powerful meta-language.

Except as an implementation architecture for meta-level inference systems, we can look at the partial-reflection interpreter defined above in a more general way. If it is true that this interpreter is indeed "adequate" (as discussed above) then a large number of meta-level interpreters can be reformulated in terms of the programmable steps of the interpreter. Usually, we think of a meta-level interpreter as an arbitrarily complex program formulated in a powerful meta-language (such as Horn Clause logic), whereas the above interpreter consists of a small number of clearly defined elementary steps. Each of the programmable steps might be an arbitrarily complex program in the meta-level language, but their behaviour is narrowly constrained, and the interpreter itself can be defined in a very small vocabulary. In fact, because all the predicates involved are total and determinate, we don't really need the full power of Horn Clause logic to define the main loop of the interpreter. The 11 predicates making up the definition of the interpreter can be seen as an elementary language which is powerful enough to define a large set of different control regimes. This would open up all kinds of possibilities for the automatic manipulation of meta-interpreters, such as proving their combinatorial soundness and completeness, automatically compiling them into a lower level language, providing uniform debugging tools for meta-level interpreters etc. All these activities are very difficult to do for general meta-level inference systems, but become possible using the uniform notation for meta-level interpreters suggested by the system described above.

Chapter 9

Conclusions and further work

In this final chapter we summarise the achievements of the previous chapters, and we draw overall conclusions from these results. As with any research, a number of questions remain unanswered, and a number of new questions arise. We discuss these open questions, and briefly describe some possible approaches to their solutions.

9.1 Summary

Chapter 3 used descriptions of a number of meta-level architectures from chapter 2 to arrive at a classification of the array of architectures reported in the literature. This classification is based on the distribution of activity between the object-level and meta-level of a system and on the communications between them. Furthermore, a number of secondary properties of meta-level architectures were distinguished, such as combinatorial completeness and soundness, the possibility for partial specifications and the linguistic relation between object-level and meta-level.

This classification of meta-level architectures allowed us to make a systematic comparison between different types of systems. This comparison led us to argue that one type of system, which we called bilingual meta-level inference systems, has a number of significant advantages over the other types of systems.

Chapter 4 took an existing theory of the essential components of any meta-level system, and applied it to logic-based bilingual meta-level inference systems in particular. This specialisation to one particular type of system allowed us to refine this theory to a much greater level of detail, resulting in a number of essential components for any system of this type. These components were then used to define more precisely some of the properties initially discussed in chapter 3.

Chapter 5 described in some detail the architecture and implementation of a particular logic-based bilingual meta-level inference system. The practical experiences with this system were reported, and a number of essential choices made in this architecture were discussed. All the components of such a system as described in the theory of chapter 4 could be recognised in the architecture of this system. The main problem hampering the practical use of this system was its run-time inefficiency, due mainly to the extra level of interpretation imposed by the meta-level interpreter.

In chapter 6 we further investigated this problem of meta-level overhead. The first half of chapter 6 gave a model of the utility of meta-level effort. This was used to show that after a certain amount of meta-level effort the gain in reduction of object-level inference will be offset by the increase in cost of meta-level inference. This behaviour was illustrated in two simple hypothetical systems with different cost/benefit curves.

The second part of chapter 6 investigated the quantitative behaviour of logic-based mono-lingual and bilingual meta-level inference systems. A comparison was made between their respective run-time overheads. Two different measures were used, one in terms of the number of logical inferences needed by the system to perform a certain task, and one in terms of the number of logical inferences per second that the system can make. Although these two measurements gave quantitatively different results, their qualitative behaviour was the same, showing that the overhead problem is indeed of significant size, and particularly so for bilingual systems.

Chapter 7 investigated partial evaluation, the main technique in the literature for solving the problem of meta-level overhead. A number of significant problems with this technique were uncovered, mainly stemming from dynamic changes to the object-level theory and the lack of static information at partial evaluation time. A number of solutions (often heuristic in nature) were suggested to solve at least some of these problems.

Chapter8 described another solution to the problem of meta-level overhead. The solution of this chapter called partial reflection, is based on the idea that a system with a meta-level inference architecture need not necessarily be implemented that way. We showed that it is possible to implement a meta-level inference system as a particular object-level inference system in such a way that none (or not many) of the advantages of a meta-level inference system are lost. By implementing a meta-level inference system as an object-level inference system, it becomes possible to implement some of the meta-level theory at a lower, and therefore more efficient level in the system.

9.2 Conclusions

The conclusions we can draw from the work summarised above are as follows:

- The extreme version of a meta-level system, namely the bilingual meta-level inference system, is the best type of meta-level architecture for realising the goal of an explicit representation of control knowledge.

- The problem of meta-level overhead is even larger than usually acknowledged. In particular for bilingual systems, the overhead is not only large but also increases with the size of the computation.

- The available technique for dealing with the problem, partial evaluation, has a number of serious problems not usually acknowledged in the literature.

- Although some of the problems with partial evaluation were overcome, and two further solutions to the overhead problem were suggested, the problem of meta-level overhead remains largely an open one, and remains the major obstruction to the use of meta-level inference systems in practical applications.

9.3 Future work

In the light of the conclusions above, the problem of reducing meta-level overhead for bilingual meta-level inference systems remains an important research problem. A number of immediate tasks arises from the work in chapters 7 - 8:

- Further work remains to be done on improving partial evaluation. In particular the exponential growth in code size needs to be tackled. Extensions to the partial evaluation algorithm as suggested by [Owen, 1988b] need further study and implementation, as does the possibility for the automatic generation of specialised stop criteria.

- Implementations are needed of the systems described in chapter8, in order to assess its behaviour on realistic examples. Our current estimates of the behaviour of this system is promising but is based on simulated measurements of LI-ratings, rather than on experiments with a proper implementation.

- Other techniques than the ones discussed in chapters 7 - 8 should be pursued. Two possible approaches that remain unexplored in this work are caching and compilation:

  Caching: A typical phenomenon in meta-level inference systems is repeated computation. Computations based on values that are fixed during the execution of the system and that are known beforehand (such as lengths of clauses or certainty values associated with rules) can be removed by partial evaluation. However, it would be an interesting approach to also store at run-time intermediate results which are based on dynamic values if they are also often repeatedly computed. Of course, these cached values can only be used again later in the computation if none of the input values for that result have changed in the mean time. This means that some form of consistency maintenance will have to be done in order to find out when a cached value will have to be recomputed. An example of such an approach applied to the control of object-oriented systems can be found in [van Marcke, 1986]. A logic-based system that uses a similar technique (although for object-level and not for meta-level computations) is the PROLEARN system in [Priedites and Mostow, 1987]. This approach should not be confused with Owen's $I_{1,Lemma}$ interpreter mentioned in section 6.4, since that interpreter stores object-level results as lemmas, whereas the caching technique would store meta-level results.

  Compilation Perhaps one of the most obvious solutions to the overhead problem is the translation of the meta-level theory into an executable language at a lower, and more efficient level. This idea is based on the observation that there is no reason why the language used to specify a control strategy has to be identical to the language used to run it. It is possible (and even likely), that the requirements for a language for the formulation of a meta-level theory by a user are quite different from the requirements for a system instruction set. However, all systems described in chapter 2 use the same language for both purposes: at execution time they interpret directly the specification as it was given by the user. It is possible to translate the specification given by the user

into something that can be executed more efficiently. An assumption that has to be made in this approach is that the control strategy is not subject to run-time changes. Give the arguments in [Reichgelt and van Harmelen, 1985, Reichgelt and van Harmelen, 1986] and [Chandrasekaran, 1983, Chandrasekaran, 1985b, Chandrasekaran, 1985a, Chandrasekaran, 1987], which claim that the control regime is a function of the type of task that is performed by the system, this is quite a reasonable assumption to make. The feasibility of this approach depends on the language that is used to formulate the meta-level theory. The Socrates system from chapter 5 allowed unrestricted use of Horn Clause logic for this purpose. For such a general purpose language, only general purpose compilation of Horn Clause logic can be used, as in general Prolog systems for instance, and the fact that the compilation concerns a meta-level theory cannot be exploited. On the other hand, a system such as described in chapter 8 defines a fixed number of predicates in the meta-level theory that are used to formulate a control regime. As a result, much more elaborate compilation could be done in such a system, since more is known about the intended meaning of these predicates. However, this is not to say that the particular predicates proposed in that chapter are the ideal set, and the problem of compilation is thus intimately connected to the choice of a meta-level language, which is an open problem in general (see below).

A number of other questions, not directly connected with the problems of meta-level overhead, have arisen from previous chapters:

- The choice of a good meta-level language for expressing control strategies remains open. Widely different languages have been used by the systems described in chapter 2, and although we chose logic as the meta-level language for the Socrates system of chapter 5, a number of reasons were given why that choice was not ideal (although there were also a number of advantages). This choice of language for the meta-level theory will influence many aspects of the system, including efficiency, expressiveness, explanation and debugging.

- We have argued in chapters 3, 4 and 5, that combinatorial soundness is a desirable property of a meta-level architecture. Yet of all the systems discussed in chapter 2, NuPRL is the only meta-level inference system which actually enforces this property, through the typing scheme employed in the meta-level language. Also, the partial reflection system from chapter 8 enforced this property through allowing a limited set of meta-level predicates. A systematic investigation of ways of enforcing this property remains to be carried out.

- A problem arising from the result of the experiments performed in chapter 6 is that LI and LI/*sec* measures were not good measures of the run time complexity of a logic program. This showed up also in the discrepancies between the quantitative results obtained using these measures. A better measure would take into account the unification complexity as well as the run-time complexity, and would count the size of search trees rather than only proof trees.

All of these problems would provide interesting topics for further research.

Appendix A

Code for a partial evaluator

```
        % Based on [Takeuchi & Furukawa, 1986]

        % The main data-structures used are:
        %       - standard lists of goals for conjunctive lists.
        %       - lists of clause-records for disjunctive lists.
        %       - a clause-record looks like:   cl(Head, Done, ToBeDone)
        %         where ToBeDone is the original body of the clause,
        %         before partial evaluation, and Done is the new body
        %         after partial evaluation.
        %         Goals get gradually moved from ToBeDone into Done.

        % peval(Goal, NewDef): Given a theory stored in the database,
        % and partial input as specified in Goal, the database theory
        % can be reformulated as NewDef.
        %
        % First clause just initialises the stack to the empty list.
        %
peval(Goal, NewDef) :- peval(Goal, NewDef, []).

        % The first three clauses are just to catch special cases:
        % - infinite-recursion,
        % - evaluable predicates, and
        % - mixed computation.
peval(Goal, inf_loop, Stack) :- loop(Goal, Stack), !.
peval(Goal, Success_Flag, _) :- evaluable(Goal,Success_Flag), !.
peval(Goal, Results, _) :- executable(Goal), execute(Goal, Results),!.

        % This is where the real work starts:
        % get all clauses from the database, convert them into
        % clause-records, and hand them on to the next layer of
        % computation:
peval(Goal, NewDef, Stack) :-
```

```
    clauses(Goal, Cls), !,
    translate(Cls, Translated),
    peval_clauses(Translated, NewDef, [Goal | Stack]).

        % peval_clauses is an intermediate layer between the top-level
        % goal and peval_clauses1 where the clauses get expanded.
        %
        % The first clauses in this intermediate layer catches goals
        % that have no defining clauses.
        %
        % The second clause passes the clauses-to-be-processed on to
        % the next layer.
peval_clauses([], fail, _) :- !.
peval_clauses(Clauses, Ans, Stack) :-
    peval_clauses1(Clauses, Temp, Stack),
    close(Temp, Ans).               % close changes [] back to fail.
                                    % This is just cosmetic.

        % peval_clauses1 performs the real work:
        % for every clause in the clause-list
        % do:   for every Goal in the body of the clause
        %       do:     peval_goal(Goal);
        %               move result of this into Done-field
        %               delete Goal from the ToBeDone-field
        %       done.
        % done.
        %
        % First clause catches "facts" in the Prolog database
        % (clauses with body true).
peval_clauses1([cl(Head, [], [true]) | Cls],
               [cl(Head, [], []    ) | Ans],
               Stack) :- !,
    peval_clauses1(Cls, Ans, Stack).
        % Second clause terminates iteration over goals in the
        % clause-body: the ToBeDone-field is [], and this final version
        % of the clause is passed on into the output argument:
peval_clauses1([cl(Head, Done, []) | Cls],
               [cl(Head, Done, []) | NTail],
               Stack) :-
    peval_clauses1(Cls, NTail, Stack).
        % Third clause terminates iteration over clauses in the
        % clause-list:
peval_clauses1([], [], _).
        % Last clause does the real work: partially evaluate the first
        % goal from the ToBeDone-field, unfold the result into
```

```
        % or-parallel clauses, and recurse on the result:
peval_clauses1([cl(Head, Done, [Goal | Rest]) | Cls], Ans, Stack) :-
    peval(Goal, NewGoals, Stack),
    unfold(NewGoals, cl(Head, Done, [Goal | Rest]), NewCls),
    append(NewCls, Cls, AllCls),
    peval_clauses1(AllCls, Ans, Stack).

        % unfold takes the result of partially evaluating a subgoal in
        % the body of a clause, and makes a new copy of the clause for
        % every possible expansion of the subgoal.
        %
        % The first clause catches infinitely recursive subgoals: The goal
        % is moved from the ToBeDone- into the Done-field without being
        % changed.
unfold(inf_loop, cl(Head, Done, [Goal | Rest]),
                [cl(Head, Done_plus_Goal, Rest)]) :-
    append(Done, [Goal], Done_plus_Goal).

        % The second clause catches unexpanded evaluable (built-in)
        % predicates: The treatment is exactly the same as for as for
        % infinitely recursive goals in the previous clause.
unfold(not_done_evaluable, cl(Head, Done, [Goal | Rest]),
                          [cl(Head, Done_plus_Goal, Rest)]) :-
    append(Done, [Goal], Done_plus_Goal).
        % Third clause catches successfully evaluated built-ins: the
        % goal disappears from the code.  (possible results will have
        % registered through the bindings of shared variables).
unfold(success_evaluable, cl(Head, Done, [_ | Rest]),
                         [cl(Head, Done, Rest)]).
        % Fourth clause catches failed built-ins: the processing of this
        % clause is aborted.
unfold(fail_evaluable, _, []).
        % Fifth clause catches failing partial evaluation. Just like
        % failed built-ins above.
unfold(fail, _, []).
    % This clause terminates the iteration in the last clause.
unfold([], _, []).
        % Last clause does the real work: for every expansion of the
        % subgoal 'Goal', we return a new copy of the clause, with Goal
        % moved from ToBeDone into Done.  We make sure that these
        % or-parallel clauses don't share variables.
unfold([cl(G, Body, []) | RestGs],
        cl(Head, Done, [G1 | Rest]),
       [cl(Head, NewDone, Rest) | RestNew]) :-
    copy(cl(Head, Done, [G1 | Rest]), NewClause),
```

```
        % We need to take a new copy here, to make sure that or-parallel
        % clauses get evaluated in independent environments.
    G = G1,
    append(Done, Body, NewDone),
    unfold(RestGs, NewClause, RestNew).

    % Translates the output of clauses/2 into the clause-record format.
    % At this stage, the ToBeDone-field is still [].
translate([(Head:-Body) | Cls], [cl(Head, [], Blst)  | Tail]) :- !,
    conjunction_to_list(Body, Blst),
    translate(Cls, Tail).
translate([], []).
```

Bibliography

[Abramson and Rogers, 1988] Abramson, H. and Rogers, M., editors (1988). *Meta-programming in logic programming*, Cambridge, Mass. MIT Press. Proceedings of the Meta'88 Workshop on meta-programming in logic programming, Bristol, June 1988. 135

[Aiello and Levi, 1984] Aiello, L. and Levi, G. (1984). The uses of metaknowledge in AI systems. In *Proceedings of the Sixth European Conference on Artificial Intelligence, ECAI '84*, pages 707–717, Pisa. Reprinted in *Meta-level architectures and reflection*, Maes, P. and Nardi, D. (eds.), North Holland Publishers, Amsterdam, 1988, pp. 243-254. 17

[Amarel, 1968] Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D., editor, *Machine Intelligence Vol. 3*, pages 131–172. Edinburgh University Press. 51

[Attardi and Simi, 1984] Attardi, G. and Simi, M. (1984). Metalanguage and reasoning across viewpoints. In *Proceedings of the Sixth European Conference on Artificial Intelligence, ECAI '84*, pages 315–324, Pisa. 66

[Batali, 1983] Batali, J. (1983). Computational introspection. Technical Report 701, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts. 65, 66

[Beetz, 1987] Beetz, M. (1987). Specifying meta-level architectures for rule-based systems. SEKI Report SR-87-06, Universität Kaiserslautern, Fachbereich Informatik, Kaiserslautern. 122

[Bjorner et al., 1987] Bjorner, D., Ershov, A., and JOnes, N., editors (1987). *Proceedings of the workshop on Partial Evaluation and Mixed Computation*, Avernaes, Denmark. 135

[Bledsoe, 1981] Bledsoe, W. (1981). Non-resolution theorem proving. *Artificial Intelligence Journal*, 9:1–35. Also in: *Readings in Artificial Intelligence* Webber, B.L. and Nilsson, N.J. (eds.), Tioga publishing, Palo Alto, CA, pp. 91-108. 75

[Bowen and Kowalski, 1982] Bowen, K. and Kowalski, R. (1982). Amalgamating language and metalanguage in logic programming. In Clark, K. and Tarnlund, S., editors, *Logic Programming*, pages 153–172. Academic Press. 46, 63

[Breuker and Wielinga, 1986] Breuker, J. and Wielinga, B. (1986). Models of expertise. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 306–318, Brighton. 17, 73, 78, 81

[Bundy, 1987] Bundy, A. (1987). What kind of field is artificial intelligence? In Partridge, D. and Wilks, Y., editors, *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, New Mexico. Also DAI Research Paper No. 305, Department of Artificial Intelligence, University of Edinburgh. 20

[Bundy, 1988] Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *Proceedings of the 9th Conference on Automated Deduction CADE9*, pages 111–120. Springer Verlag. Also as DAI Research Paper No. 349, Department of Artificial Intelligence, University of Edinburgh. 41

[Bundy et al., 1979] Bundy, A., Byrd, L., Luger, G., Mellish, Milne, C., R., and Palmer, M. (1979). Solving mechanics problems using meta-level inference. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI '79*, pages 50–64, Tokyo. Also in: Expert Systems in the Micro Electronic Age, Michie, D. (ed.), Edinburgh University Press, 1979. 37

[Bundy and Sterling, 1988] Bundy, A. and Sterling, L. (1988). Meta-level inference two applications. *Journal of Automated Reasoning*, 4(1):15–28. Also: DAI Research Paper No. 273, Department of Artificial University of Edinburgh, 1985, entitled "Meta-level Inference in Algebra", (revised version of DAI Research Paper No. 164). Also in: *Proceedings of the Capri-85 Conference on AI*, North Holland Publishers, Amsterdam. 35

[Bundy et al., 1988] Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1988). Experiments with proof plans for induction. DAI Research Paper 413, Department of Artificial Intelligence, University of Edinburgh. To appear in *Journal of Automated Reasoning*. 41

[Bundy and Welham, 1981] Bundy, A. and Welham, B. (1981). Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence Journal*, 16(2):189–212. 17, 35

[Chan and Wallace, 1988] Chan, D. and Wallace, M. (1988). A treatment of negation during partial evaluation. In Abramson, H. and Rogers, M., editors, *Meta-programming in logic programming*, pages 299–318, Cambridge, Mass. MIT Press. Proceedings of the Meta'88 Workshop on meta-programming in logic programming, Bristol, June 1988. 115, 135

[Chandrasekaran, 1983] Chandrasekaran, B. (1983). Towards a taxonomy of problem-solving types. *AI Magazine*, 4:9–17. 62, 73, 77, 78, 81, 160

[Chandrasekaran, 1985a] Chandrasekaran, B. (1985a). Expert systems: Matching techniques to tasks. In Reitman, W., editor, *Artificial Intelligence Applications for Business*. Ablex Corp. 62, 73, 77, 78, 81, 160

[Chandrasekaran, 1985b] Chandrasekaran, B. (1985b). Generic tasks in expert system design and their role in explanation of problem solving. In *NAS/ONR Workshop on AI and distributed problem solving*. 62, 73, 77, 78, 81, 160

[Chandrasekaran, 1987] Chandrasekaran, B. (1987). Towards a functional architecture for intelligence based on generic information processing tasks. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence, IJCAI '87*, pages 1183–1192, Milan. 62, 73, 77, 78, 81, 160

[Clancey, 1983a] Clancey, W. (1983a). The advantages of abstract control knowledge in expert system design. In *Proceedings of the Third Annual Meeting of the American Association for Artificial Intelligence, AAAI '83*, pages 74–78, Washington, DC. 15, 17

[Clancey, 1983b] Clancey, W. (1983b). The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence Journal*, 20:215–251. Also: Stanford Heuristic Programming Project, Memo HPP-81-17, 1981, also numbered STAN-CS-81-896. 17, 26, 34, 131

[Clancey, 1985] Clancey, W. (1985). Representing control knowledge as abstract tasks and metarules. In Coombs, M. and Bolc, L., editors, *Computer Expert Systems*. Springer Verlag. Also: Stanford Knowledge Systems Laboratory, Working Paper No. KSL-85-16. 15, 17

[Clancey and Bock, 1982] Clancey, W. and Bock, C. (1982). MRS/NEOMYCIN: Representing metacontrol in predicate calculus. Technical Report HPP-82-31, Stanford Heuristic Programming Project Report. 34

[Clancey and Letsinger, 1981] Clancey, W. and Letsinger, R. (1981). NEOMYCIN: Reconfiguring a rule-based expert system for application to teaching. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI '81*, pages 829–836, Vancouver. Also: Stanford Heuristic Programming Project, Memo HPP-81-2, May 1982, also numbered STAN-CS-82-908. 17, 34

[Clark and McCabe, 1982] Clark, K. and McCabe, F. (1982). IC-prolog language features. In Clark, K. and Tarnlund, S., editors, *Logic Programming*, pages 253–266. Academic Press. 43

[Constable et al., 1986] Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., and Smith, S. (1986). *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall. 39

[Corlett et al., 1988] Corlett, R., Davies, N., Khan, R., Reichgelt, H., and van Harmelen, F. (1988). Socrates: A flexible toolkit for building logic based expert systems. In Jackson, P., Reichgelt, H., and van Harmelen, F., editors, *Logic-Based Knowledge Representation*, pages 132–142. MIT Press, Cambridge Massachusetts. A shorter version published in: *The Knowledge-Based Systems Journal*, Vol. **1**, No. 3. 73, 74, 84

[Coscai et al., 1988] Coscai, P., Franceschi, P., Levi, G., Sardu, G., and Torre, L. (1988). Meta-level definition and compilation of inference engines in the epsilon logic programming environment. In Kowalski, R. and Bowen, K., editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 359–373, Seattle. 121

[Davies, 1988] Davies, N. (1988). Combining default reasoning and first order logic. In *Proceedings of the Eighth Technical Conference of the British Computer Society Specialist Group on Expert Systems, Expert Systems '88*, Brighton. 82

[Davis, 1978] Davis, R. (1978). Knowledge acquisition in rule based systems: Knowledge about representations as a basis for system construction and maintenance. In D.A., W. and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems, Waterman*, pages 99–134. Academic Press, New York. 18

[Davis, 1980] Davis, R. (1980). Meta-rules: Reasoning about control. *Artificial Intelligence Journal*, 15:179–222. 15, 17, 25

[Davis, 1982] Davis, R. (1982). TEIRESIAS: Applications of meta-level knowledge. In Davis, R. and Lenat, D., editors, *Knowledge-Based Systems in Artificial Intelligence*, pages 227–490. McGraw-Hill, New York. 25

[Davis and Buchanan, 1979] Davis, R. and Buchanan, B. (1979). Meta-level knowledge: Overview and applications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence, IJCAI '77*, pages 920–927, Cambridge, Mass. Also in: *Readings in Knowledge Representation*, Brachman, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 389-398. 17, 18

[des Rivières and Smith, 1984] des Rivières, J. and Smith, B. (1984). The implementation of procedurally reflective languages. In *Proceedings of the ACM Symposium on LISP and functional programming*, pages 331–347, Austin, Texas. 48, 49, 50

[Devanbu et al., 1986] Devanbu, P., Freeland, M., and Naqvi, S. (1986). A procedural approach to search control in prolog. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 53–57, Brighton. 46

[Dincbas and Le Pape, 1984] Dincbas, M. and Le Pape, J. (1984). Metacontrol of logic programs in METALOG. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 361–370, Tokyo. 46

[Erman et al., 1981] Erman, L., London, P., and Fickas, S. (1981). The design and an example use of Hearsay-III. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI '81*, pages 409–415, Vancouver. 29

[Erman et al., 1984] Erman, L., Scott, A., and London, P. (1984). Separating and integrating control in a rule-based tool. In *Proceedings of the IEEE workshop on principles of knowledge based systems*, pages 37–43, Denver, Colorado. 15, 26

[Ershov, 1982] Ershov, A. (1982). Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18:41–67. 115

[Eshghi, 1986] Eshghi, K. (1986). *Meta-Language in Logic Programming*. PhD thesis, Department of Computing, Imperial College of Science and Technology, London. 46, 153

[Feferman, 1962] Feferman, S. (1962). Transfinite recursive progressions of axiomatic theories. *The Journal of Symbolic Logic*, 27(3):259–316. 47, 59

[Futamura, 1971] Futamura, Y. (1971). Partial evaluation of computation process- an approach to a compiler-compiler. *Journal of Systems, Computers, Controls*, 2(5):45–50. 115

[Gallagher, 1986] Gallagher, J. (1986). Transforming logic programs by specialising interpreters. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, Brighton. 115

[Gallaire and Lasserre, 1979] Gallaire, M. and Lasserre, C. (1979). Controlling knowledge deduction in a declarative approach. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI '79*, pages S1–S6, Tokyo. 45

[Gallaire and Lasserre, 1982] Gallaire, M. and Lasserre, C. (1982). Meta-level control for logic programs. In Clark, K. and Tarnlund, S., editors, *Logic Programming*, pages 173–188. Academic Press. 44, 153

[Genesereth et al., 1980] Genesereth, M., Greiner, R., and Smith, D. (1980). MRS manual. Memo HPP-80-24, Stanford Heuristic Programming Project. 30

[Genesereth and Smith, 1982] Genesereth, M. and Smith, D. (1982). Meta-level architecture. Memo HPP-81-6, Stanford Heuristic Programming Project. 30

[Genesereth and Smith, 1983] Genesereth, M. and Smith, D. (1983). An overview of meta-level architecture. In *Proceedings of the third Annual Meeting of the American Association for Artificial Intelligence, AAAI '83*, pages 119–124, Washington, DC. 30

[Gordon et al., 1979] Gordon, M., Milner, R., and Wadsworth, C. (1979). *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag. 40, 87, 90

[Hayes, 1973] Hayes, P. (1973). Computation and deduction. In *Proceedings of the Symposium on the Mathematical Foundations of Computer Science*, pages 105–117. Czechoslovakian Academy of Sciences. 38

[Hayes, 1974] Hayes, P. (1974). The language GOLUX. Technical report, University of Essex. 38

[Hayes, 1977] Hayes, P. (1977). In defense of logic. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence, IJCAI '77*, pages 559–565, Cambridge, Mass. 19

[Hayes-Roth, 1984] Hayes-Roth, B. (1984). BB1: an architecture for blackboard systems that control, explain and learn about their own behaviour. Technical Report 84-16, Stanford Heuristic Programming Project. 29

[Hayes-Roth, 1985] Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence Journal*, 26:251–321. 29

[Hayes-Roth et al., 1983] Hayes-Roth, F., Waterman, D., and Lenat, D. e. (1983). *Building Expert Systems*. Addison-Wesley. 18

[Hill and Lloyd, 1988] Hill, P. and Lloyd, J. (1988). Analysis of meta-programs. In Abramson, H. and Rogers, M., editors, *Meta-programming in logic programming*, pages 23–52, Cambridge, Mass. MIT Press. Also (a longer version): Technical Report CS-88-08, Department of Computer Science, University of Bristol. 62

[Hogger, 1984] Hogger, C. (1984). *Introduction to Logic Programming*, volume 21 of *APIC Studies in Data Processing*. Academic Press. 70

[Horn, 1951] Horn, A. (1951). On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16:14–21. 77

[Hudlicka and Lesser, 1984] Hudlicka, E. and Lesser, V. (1984). Meta level control through fault detection and diagnosis. In *Proceedings of the fourth Annual Meeting of the American Association for Artificial Intelligence, AAAI '84*, pages 153–161, Austin, Texas. 38

[Jackson et al., 1989] Jackson, P., Reichgelt, H., and van Harmelen, F. (1989). *Logic-based knowledge representation*. Logic Programming. MIT Press, Cambridge, Mass. 81, 84

[Jansweijer et al., 1986] Jansweijer, W., Elshout, J., and Wielinga, B. (1986). The expertise of novice problem solvers. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 576–585, Brighton. 37

[Kleene, 1952] Kleene, S. (1952). *Introduction to Metamathematics*. Van Nostrand, New York. 116

[Komorowski, 1982] Komorowski, H. (1982). Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of prolog. In *Proceedings of the ninth conference on the Principles of Programming Languages, POPL'82*, pages 225–267, Albuquerque, New Mexico. 115

[Kowalski, 1979] Kowalski, R. (1979). *Logic for Problem Solving*. Artificial Intelligence Series. North Holland Publishers, Amsterdam. 19, 101, 146

[Kramer, 1984] Kramer, B. (1984). Representing control strategies using reflection. In *Proc. Fourth Conference of the Canadian Society for Computational Studies of Intelligence, CSCSI-84*, London, Ontario. 46

[Levesque, 1984] Levesque, H. (1984). A fundamental tradeoff in knowledge representation and reasoning. In *Proc. Fourth Conference of the Canadian Society for Computational Studies of Intelligence, CSCSI-84*, pages 141–152, Ontario. Revised version in *Readings in knowledge representation*, Brachman, R. and Levesque, H. (eds.), pp. 41-70, Morgan Kaufmann, Los Altos, CA, 1985. 20

[Levi, 1988] Levi, G. (1988). Object level reflection of inference rules by partial evaluation. In Maes, P. and Nardi, D., editors, *Meta-level architectures and reflection*, pages 313–328. North Holland Publishers, Amsterdam. 115

[Lloyd, 1984] Lloyd, J. (1984). *Foundations of Logic Programming.* Symbolic Computation Series. Springer Verlag, Berlin. 135

[Lloyd, 1988] Lloyd, J. (1988). Directions for meta-programming. In *Proceedings of the the conference on Fifth Generation Computer Systems, FGCS'88*, Tokyo. Also: Technical Report CS-88-10, Department of Computer Science, University of Bristol, 1988. 62

[Lloyd and Shepherdson, 1987] Lloyd, J. and Shepherdson, J. (1987). Partial evaluation in logic programming. Technical report CS-87-09, Department of Computer Science, University of Bristol. 135

[Lowe, 1988] Lowe, H. (1988). Empirical evaluation of meta-level interpreters. Master's thesis, Department of Artificial Intelligence, University of Edinburgh. (M.Sc. Thesis). 113

[Maes, 1986a] Maes, P. (1986a). Introspection in knowledge representation. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 256–269, Brighton. 32, 65, 66

[Maes, 1986b] Maes, P. (1986b). Reflection in an object-oriented language. AI-Laboratory Memo 86-8, Vrije Universiteit Brussel. Also in: Background papers for the Insight Working Week 5, Dryburgh Abbey, Scotland. 32, 65, 66

[Maes, 1987] Maes, P. (1987). Computational reflection. AI Laboratory, Technical Report 87-2, Vrije Universiteit Brussel. 49

[Martin-Löf, 1982] Martin-Löf (1982). Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Amsterdam. North-Holland. 39

[Martin-Löf, 1973] Martin-Löf, P. (1973). An intuitionistic theory of types: predicative part. In Rose, H. and (eds.), J. S., editors, *Logic Colloquium'73*, pages 73–118, Amsterdam. North-Holland. 39

[Meltzer, 1971] Meltzer, B. (1971). Prolegomena to a theory of efficiency of proof procedures. In *Artificial Intelligence and Heuristic Programming*, pages 15–33. Edinburgh University Press. Department of Computational Logic, Memo DCL-37, University of Edinburgh, 1971. 68

[Mitchell et al., 1986] Mitchell, T., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80. Also: Tech. Report ML-TR-2, SUNJ Rutgers, 1985. 115, 129

[Moore, 1982] Moore, R. (1982). The role of logic in knowledge representation and commonsense reasoning. In *Proceedings of the second Annual Meeting of the American Association for Artificial Intelligence, AAAI '82*, pages 428–433, Pittsburgh, PA. Also in: *Readings in Knowledge Representation*, Brachman, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 335-342. 20

[Moore, 1984] Moore, R. (1984). The role of logic in artificial intelligence. AI Centre Technical Note 335, SRI. 19

[Nilsson, 1983] Nilsson, M. (1983). FOOLOG - a small and efficient prolog interpreter. UPMAIL Technical Report 20, Computing Science Department, Uppsala University, Sweden. 84

[Nilsson, 1984] Nilsson, M. (1984). The world's shortest prolog interpreter? In Campbell, J., editor, *Implementations of Prolog*, pages 87–92. Ellis Horwood. 84

[O'Keefe, 1985] O'Keefe, R. (1985). On the treatment of cuts in prolog source-level tools. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 68–72, Boston. 101

[O'Keefe, 1988] O'Keefe, R. (1988). Practical prolog for real programmers. Tutorial No. 8 of the Fifth Internation Conference and Symposium on Logic Programming, University of Washington, Seattle, Washington. 113

[Owen, 1988a] Owen, S. (1988a). The development of explicit interpreters and transformers to control reasoning about protein topology. Technical Memo HPL-ISC-TM-88-015, Hewlett-Packard Laboratories Bristol Research Centre, Bristol, U.K. 113

[Owen, 1988b] Owen, S. (1988b). Issues in the partial evaluation of meta-interpreters. In Abramson, H. and Rogers, M., editors, *Meta-programming in logic programming*, pages 319–340, Cambridge, Mass. MIT Press. Proceedings of the Meta'88 Workshop on meta-programming in logic-programming, Bristol, June 1988. 115, 135, 159

[Prawitz, 1965] Prawitz, D. (1965). Natural deduction, a proof-theoretical study. *Stockholm Studies in Philosophy*, 3:5–47. 75

[Priedites and Mostow, 1987] Priedites, A. and Mostow, J. (1987). PROLEARN: towards a prolog interpreter that learns. In *Proceedings of the Sixth Annual Meeting of the American Association for Artificial Intelligence, AAAI '87*, Seattle, Washington. Also: Research Report No. ML-TR-13, Lab. for Computer Science Research, Hill Center for Mathematical Sciences, Rutgers University, New Brunswick, New Jersey 08903. 133, 159

[Reichgelt, 1987] Reichgelt, H. (1987). Semantics for a reified temporal logic. In *AISB '87*, pages 49–62, Edinburgh. Also: DAI Research Paper No. 299, Department of Artificial Intelligence, University of Edinburgh, 1987. 82

[Reichgelt and van Harmelen, 1985] Reichgelt, H. and van Harmelen, F. (1985). Relevant criteria for choosing an inference engine in expert systems. In *Proceedings of the fifth Technical Conference of the British Computer Society Specialist Group on Expert Systems, Expert Systems '85*, pages 21–30, Warwick. Also: DAI Research Paper No. 270, Department of Artificial Intelligence, University of Edinburgh, 1987. 19, 62, 73, 77, 121, 160

[Reichgelt and van Harmelen, 1986] Reichgelt, H. and van Harmelen, F. (1986). Criteria for choosing representation languages and control regimes for expert system. *The Knowledge Engineering Review*, 1(4):2–17. Also: DAI Research Paper No. 287, Department of Artificial Intelligence, University of Edinburgh, 1987. 19, 62, 73, 77, 80, 81, 121, 160

[Reichgelt and van Harmelen, 1987] Reichgelt, H. and van Harmelen, F. (1987). Building expert systems using logic and meta-level interpretation. DAI Research Paper 303, Department of Artificial Intelligence, University of Edinburgh. 73

[Rosenschein and Singh, 1983] Rosenschein, J. and Singh, V. (1983). The utility of meta-level effort. Report HPP-83-20, Stanford Heuristic Programming Project. 94

[Safra and Shapiro, 1986] Safra, S. and Shapiro, E. (1986). Meta interpreters for real. In *Proceedings of IFIPS '86*, pages 271–278, Dublin. Also: Report No. CS86-11, Department of Computer Science, The Weizmann Institute of Science, 1986. Reprinted in: Concurrent Prolog Collect Papers, E. Shapiro (ed.), MIT Press, 1987, Vol II, 166-179. 115

[Shapiro, 1983] Shapiro, E. (1983). Logic programs with uncertainties, a tool for implementing rule-based systems,. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, IJCAI '83*, pages 529–532, Karlsruhe. 85

[Shortliffe, 1976] Shortliffe, E. (1976). *Computer-Based Medical Consultations: MYCIN*. Elsevier, New York. 34

[Silver, 1986] Silver, B. (1986). *Meta-level Inference*. Studies in Computer Science and Artificial Intelligence. North Holland Publishers, Amsterdam. 35, 37, 57

[Smith, 1984] Smith, B. (1984). Reflection and semantics in LISP. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages, POPL '84*, pages 23–35, Salt Lake City, Utah. Also: Xerox PARC Intelligent Systems Laboratory Technical Report ISL-5. 48

[Smith, 1985] Smith, B. (1985). Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts. Also in: *Readings in Knowledge Representation Brachman*, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, , pp. 31-40. 19, 65

[Steele and Sussman, 1978] Steele, G. and Sussman, G. (1978). The art of the interpreter, or, the modularity complex (parts zero, one and two). Memo AIM-453, M.I.T. AI Laboratory, Cambridge, Massachusetts. 48

[Steels, 1985] Steels, L. (1985). The KRS concept system. Technical Report 86-1, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Belgium. 32

[Sterling, 1982] Sterling, L. (1982). IMPRESS - meta-level concepts in theorem proving. DAI Working Paper 119, Department of Artificial Intelligence, University of Edinburgh. 37

[Sterling, 1984] Sterling, L. (1984). Implementing problem-solving strategies using the meta-level. DAI Research Paper 209, Department of Artificial Intelligence, University of Edinburgh. 36

[Sterling and Beer, 1986] Sterling, L. and Beer, R. (1986). Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proceedings of the 3rd Symposium on Logic Programming*, pages 20–27, Salt Lake City, Utah. 108, 122, 123

[Sterling et al., 1982] Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B. (1982). Solving symbolic equations with PRESS. In Calmet, J., editor, *Computer Algebra*. Lecture Notes in Computer Science, Vol. 144, Springer Verlag. Also: DAI Research Paper No. 171, Department of Artificial Intelligence, University of Edinburgh, 1982. 35

[Sterling and Shapiro, 1986] Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. Logic Programming Series. MIT Press, Cambridge, Massachusetts. 101, 145

[Takeuchi and Furukawa, 1986] Takeuchi, A. and Furukawa, K. (1986). Partial evaluation of prolog programs and its application to meta programming. In *Proceedings of IFIPS '86*, pages 415–420, Dublin. Also: ICOT Research Centre, Technical Report TR-125, Tokyo, 1985. 115, 121

[Takewaki et al., 1985] Takewaki, T., Takeuchi, A., Kunifuji, S., and Furukawa, K. (1985). Application of partial evaluation to the algebraic manipulation system and its evaluation. Technical Report TR-148, ICOT Research Centre, Tokyo. 37, 115

[Tarnlund, 1977] Tarnlund, S. (1977). Horn clause computability. *BIT*, 17:215–226. 125

[Tarski, 1956] Tarski, A. (1956). The concept of truth in formalized languages. In *Logic, Semantics, Metamathematics*, pages 152–278. Clarendon Press, Oxford. 88

[Treur, 1988] Treur, J. (1988). Completeness and definability in diagnostic expert systems. In *Proceedings of the Eighth European Conference on Artificial Intelligence, ECAI '88*, pages 619–624, Munich. 121

[van Harmelen and Bundy, 1988] van Harmelen, F. and Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence Journal*, 30(3):401–412. Also: DAI Research Paper No. 347, Department of Artificial Intelligence, University of Edinburgh, 1987. 115, 116, 129

[van Marcke, 1986] van Marcke, K. (1986). FPPD: A consistency maintenance system based on forward propagation of proposition denials. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 278–290, Brighton. 159

[Venken, 1984] Venken, R. (1984). A prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-imisation. In *Proceedings of the Sixth European Conference on Artificial Intelligence, ECAI '84*, pages 91–100, Pisa. 115, 125

[Wallen, 1983] Wallen, L. (1983). Towards the provision of a natural mechanism for expressing domain specific global strategies in general purpose theorem-provers. DAI Research Paper 202, Department of Artificial Intelligence, University of Edinburgh. 59

[Walther, 1984] Walther, C. (1984). A mechanical solution of schubert's steamroller by many-sorted resolution. In *Proceedings of the fourth Annual Meeting of the American Association for Artificial Intelligence, AAAI '84*, pages 330–334, Austin, Texas. 82

[Waltzman, 1983] Waltzman, R. (1983). OPS5 tutorial. Technical report, Teknowledge Inc. 26

[Warner Hasling, 1983] Warner Hasling, D. (1983). Abstract explanations of strategy in a diagnostic consultation system. In *Proceedings of the third Annual Meeting of the American Association for Artificial Intelligence, AAAI '83*, pages 157–161, Washington, DC. 17

[Warner Hasling et al., 1984] Warner Hasling, D., Clancey, W., and Rennels, G. (1984). Strategic explanations for a diagnostic consultation system. *International Journal of Man-Machine Studies*, 20(1):3–19. Also: Stanford Heuristic Programming Project, Memo HPP-83-41, 1983, also numbered STAN-CS-83-996. 17, 62

[Welham, 1986] Welham, B. (1986). Declaratively programmable interpreters and meta-level inferences. Technical Memo HPL-BRC-TM-86-027, Hewlett-Packard Laboratories Bristol Research Centre. *Meta-level architectures and reflection*, Maes, P. and Nardi, D. (eds.), North Holland Publishers, 1988. 52

[Weyhrauch, 1981] Weyhrauch, R. (1981). Prolegomena to a theory of mechanised formal reasoning. *Artificial Intelligence Journal*, 13:133–170. Also in: *Readings in Artificial Intelligence*, Webber, B.L. and Nilsson, N.J. (eds.), Tioga publishing, Palo Alto, CA, , pp. 173-191. Also in: *Readings in Knowledge Representation Brachman*, R.J. and Levesque, H.J. (eds.), Morgan Kaufman, California, 1985, pp. 309-328. 47, 59, 78

Index