Coupled Resolution Engines for Programming

Knowledge Based Systems in Logic

Hamish Taylor

Doctor of Philosophy Degree

Heriot-Watt University

Computer Science Department

December 1989

© Hamish Taylor

This copy of the thesis has been supplied on condition that anyone, who consults it, is understood to recognise that the copyright rests with its author, and that any quotation from the thesis and any information derived from it may only be published without the prior written consent of the author, if a full acknowledgement of this source is made.
Declaration

The work presented in this dissertation was carried out by myself, except where due acknowledgement is made. This dissertation has not been submitted by me for any other degree at this or any other university. However, some material presented in this document has been or will be published as follows:


For

Yiya Yang
Abstract

Research on a fifth generation of computer systems aims at building high performance knowledge processing engines on the basis of the highly parallel execution of logic programs. However, outstanding problems in the design and implementation of logic programming languages make it problematic to combine parallel execution, adequate support for deduction and a systems programming capability. This combination is important for programming multi-user knowledge based systems in logic.

This research describes the design and use of a logic programming system that reconciles these three different requirements for this type of application. The system couples a concurrent logic programming or CLP engine with a multi-threaded Prolog engine. Concurrency among coupled CLP and Prolog computations is sustained partly by fine-grained and-parallelism in the CLP engine, and partly by coarse grained concurrent processing of multiple Prolog processes and a CLP engine under Unix. Communication among distinct computations is realised by message passing, which allows the architecture to reduce contention for shared memory during multi-processing.

The CLP engine sustains the process interpretation of systems programming, which enables it to execute the management system of a multi-user knowledge based system. The Prolog engine can exhibit don’t know non-determinism, which enables it to perform exhaustive searches over a knowledge base. The coupled system allows Prolog computation threads to share or to have private databases, and is compatible with scale enhancing approaches to resolution using knowledge base machines. Only minor extensions to CLP and Prolog engines are needed to realise the coupled architecture. This leaves it free to evolve with emerging CLP and Prolog implementation technology, especially in the area of multi-processing CLP languages in and-parallel and Prolog in or-parallel.

Keywords: logic programming, parallelism, deduction, control, systems programming, knowledge base
Acknowledgements

First of all I would like to thank my supervisor, Professor M.H. Williams, for creating the possibility for this research, for all his cooperation and help during this research, and for advice and assistance in writing it up and presenting it.

I would like to thank Paul Massey, Kam Fai Wong and David Ferbrache, with whom I collaborated closely on the Prolog Database Machine project, for much stimulation, fruitful argument and friendship, which helped a lot to make it all possible. I would also like to thank members of the Imperial College Parlog group, particularly Jim Crammond and Alastair Burt for discussions, arguments and help that were very valuable in carrying this research forward. I would also like to thank my colleagues at Heriot-Watt university for their support.

My father’s work in proof reading this dissertation was much appreciated, as were his many helpful suggestions that have made it more readable and understandable. Finally, but most importantly I would like to thank my wife Yiya for her careful proof-reading, encouragement, support and love. Without her this dissertation would not exist.

Much of this research was done as part of Alvey project IKBS 90 Parlog on Parallel Architectures, which was funded by the Science and Engineering Research Council with International Computers Limited as industrial collaborator.
Table of Contents

Preface ........................................................................................................................................... i
Declaration .................................................................................................................................... ii
Dedication ................................................................................................................................. iii
Abstract ...................................................................................................................................... iv
Acknowledgements .................................................................................................................. v
Table of Contents ....................................................................................................................... vi
Figures ......................................................................................................................................... xii
Tables ......................................................................................................................................... xii

Introduction ................................................................................................................................. 1

1 Fifth Generation Computer Systems ................................................................................ 1
   1.1 Knowledge Processing ........................................................................................ 3
   1.2 Declarative Programming ................................................................................... 3
   1.3 Automated Problem Solving ............................................................................... 4
   1.4 Highly Parallel Execution ................................................................................... 5
   1.5 Formal Methods .................................................................................................. 5
   1.6 Meta-Programming ............................................................................................. 6
   1.7 Database Query Capability .................................................................................. 7
   1.8 Knowledge Base Programming ........................................................................... 7
2 Research Context .................................................................................................................. 8
3 Research Objectives .............................................................................................................. 9
4 Research Contributions ........................................................................................................ 10
5 Dissertation Outline ............................................................................................................. 10

Chapter 1 Programming Knowledge Based Systems in Logic .................................................. 12
   1.1 Introduction ........................................................................................................... 12
   1.2 Logic ...................................................................................................................... 12
   1.3 Proof by Refutation .............................................................................................. 14
   1.4 Logic Programming .............................................................................................. 16
4.5 The Common Linguistic Denominator ................................................................. 101
  4.5.1 Mode Declarations ......................................................................................... 102
  4.5.2 Passive Part Concurrency ........................................................................... 102
  4.5.3 And-Sequential Operators ........................................................................... 103
  4.5.4 Or-Sequential Search ................................................................................. 105
  4.5.5 Rule of Sequencing ..................................................................................... 106
  4.5.6 Rule of Synchronisation .............................................................................. 106
    4.5.6.1 Localisation ....................................................................................... 108
    4.5.6.2 General Localised Suspension ......................................................... 111
    4.5.6.3 A Localised Suspension Algorithm ................................................. 112
    4.5.6.4 Adequacy of Localised Suspension ................................................. 114
4.6 Example Translations ....................................................................................... 115
4.7 Translating back from the Lingua Franca ....................................................... 116
4.8 The Lingua Franca .............................................................................................. 117
4.9 Conclusion .......................................................................................................... 119

Chapter 5 Aspects of the Design of a CLP Engine ................................................... 121
5.1 Introduction ........................................................................................................ 121
5.2 A CLP Execution Model .................................................................................... 121
  5.2.1 Goal-Clause Execution Model .................................................................... 122
  5.2.2 Process States ............................................................................................ 123
  5.2.3 Roles of Process Types ............................................................................. 124
  5.2.4 Lazy Clause Process Creation .................................................................. 126
  5.2.5 Process Promotion ................................................................................... 129
5.3 Memory Management Issues ........................................................................... 135
  5.3.1 Representing Processes .......................................................................... 135
    5.3.1.1 Meta-call Process ........................................................................... 137
    5.3.1.2 Goal Process .................................................................................. 138
    5.3.1.3 Clause Process .............................................................................. 141
  5.3.2 The Data Storage Model ......................................................................... 142
    5.3.2.1 Terms ............................................................................................ 142
    5.3.2.2 Records .......................................................................................... 144
5.4 Scheduling .......................................................................................................... 147
Chapter 6 Escape Exception Handling in Prolog .............................................. 151
  6.1 Introduction .................................................................................................. 151
  6.2 Exceptions and Choice Points ....................................................................... 151
  6.3 Exception Handling ....................................................................................... 155
    6.3.1 Escape Exceptions .............................................................................. 156
    6.3.1 Prolog and Escape Exceptions ......................................................... 157
  6.4 Recovery ....................................................................................................... 160
    6.4.1 Recovery without Side Effects ............................................................ 161
    6.4.2 Database Rollback ............................................................................. 162
    6.4.3 Transaction Control .......................................................................... 163
  6.5 Error Handling ............................................................................................. 165
    6.5.1 Terminating Recursion ....................................................................... 166
    6.5.2 Robust Systems .................................................................................. 168
  6.6 Implementing Exceptional Choice Points ................................................... 169
  6.7 Conclusion .................................................................................................... 172

Chapter 7 Programming Coupled Resolution Engines ......................................... 173
  7.1 Introduction .................................................................................................. 173
  7.2 Requirements of a Knowledge Based System ............................................ 173
  7.3 Meta-Interpreters ....................................................................................... 175
    7.3.1 A Lingua Franca Meta-Interpreter .................................................... 175
    7.3.2 A Prolog Meta-Interpreter ................................................................. 176
    7.3.2 A Coupled CLP-Prolog Meta-Interpreter .......................................... 177
  7.4 Concurrent Resolution .................................................................................. 179
    7.4.1 A Multi-User Prolog System ............................................................... 179
    7.4.2 Controlling Resolution Engines ....................................................... 181
    7.4.3 Multiple Solutions Predicates ............................................................ 183
  7.5 A Prolog Database Machine ........................................................................ 187
    7.5.1 Database Machine Components ........................................................ 188
Figures

Figure 1.1 Basic Logic Programming Scheme ................................................................. 17
Figure 4.1 Localised Suspension Algorithm ................................................................. 113
Figure 5.1 Lazy Creation of Clause Processes ............................................................. 127
Figure 5.2 Committed Choice Reduction of a Goal ..................................................... 130
Figure 5.3 Delayed Promotion Scheme ....................................................................... 132
Figure 5.4 Suspended Processes ................................................................................ 146
Figure 7.1 Possible Configuration of Prolog Database Machine ................................. 190

Tables

Table 1.1 Discoveries of Properties of First Order Logic .......................................... 13
Table 5.1 Tagged Data Structures ............................................................................. 143
Table 6.1 Built in Escape Exceptions ........................................................................ 165
Table 8.1 Languages for Programming Systems and Search Problems in Logic ....... 198


Introduction

This introduction outlines the context of this dissertation’s research, describes the problems it addresses and sketches the plan of the rest of this text. It begins with the motivations and themes of the Japanese Fifth Generation Computer Systems or FGCS project, because they represent a particularly clear statement of the intellectual framework for the work reported here. The research undertaken by the FGCS project has not been exclusive to Japan, but is also being carried out in universities and research centres in Britain, Sweden, U.S.A., Australia, France and elsewhere. This international research programme aims to design and build the next generation of knowledge processing computers based on logic programming. Fifth generation computer systems research has had significant successes in developing new theory, new applications areas, new programming languages, new programming techniques, new implementation strategies, and new designs for high performance knowledge processing machines. However, difficulties remain with the tactics being developed for using logic programming languages to program knowledge based systems which make it problematic for a logic programming system to combine in an efficient manner

- a complete deduction capability over knowledge bases
- a systems programming capability
- suitability for parallel execution

This dissertation addresses this research problem by describing a logic programming architecture for reconciling these different requirements. It introduces this problem by describing central ideas of the fifth generation computer systems research programme, in order to show the way in which logic programming implementations aim to contribute to the construction of knowledge based systems.

1. Fifth Generation Computer Systems

The Japanese government launched the Fifth Generation Computer Systems Project in 1982 as a ten year programme to help its industry foster more creative computer technology and to make Japan play a leading world role in that technology’s development. Two particular concerns were to open up a new market in computers to keep the Japanese computer industry growing and to improve the low productivity of current
techniques of software development. The FGCS project took as its objective the development of a new generation of computer systems oriented towards processing knowledge intensive information [Fuchi 1983]. The FGCS project envisaged these machines as very high performance knowledge processing engines based upon novel non-Von Neumann architectures which would be highly parallel and realised in VLSI using emerging fabrication technologies. They would have their own problem solving capabilities, be able to manage very large amounts of knowledge, and have intelligent interfaces which could accept spoken natural language and visual data as input and synthesise speech as output. Various types of system were envisaged including a personal workstation, a super inference machine, a knowledge base machine and a fifth generation network architecture [Moto-Oka 1982].

Logic programming was selected as the computing paradigm to realise a high level, high performance knowledge processing capability. Thus the objective of the Fifth Generation Computer Systems project crystallised into the general plan of developing special purpose computer systems which would execute logic programs in a highly parallel fashion [Fuchi & Furukawa 1986]. Other aspects of the original plan became subsidiary to this goal. Logic programming became the heart of the project as the missing link connecting knowledge information processing and highly parallel architectures which would unify capabilities in artificial intelligence, computer architecture, database semantics, linguistics, programming language semantics, and software engineering [Kowalski 1982]. Logic programming realises these aspirations of fifth generation computer systems research for a number of related reasons. It enables

- intensive knowledge processing
- high level descriptions of problems in a declarative fashion
- built in problem solving using a theorem prover
- opportunity for a high degree of parallel execution
- suitability for the application of formal methods
- significant meta-programming capability
- a synthesis of programming with database query languages
- rapid programming of knowledge based systems

Each of these issues is worth amplifying to see the place of logic programming in fifth generation computer systems research.
1.1. Knowledge Processing

The scope and demand for computational solutions to problems have been rapidly expanding. New applications in natural language processing, computer aided design, computer aided manufacture, decision support systems, deductive databases, expert systems, intelligent interfaces, and tutorial systems have been creating an increasing demand for computational capabilities not well suited to the numerical and data processing style at which the von Neumann computer excels. A common factor underlying many of these new applications is a demand to process knowledge. Processing knowledge involves operations like acquiring, revising, storing, regimenting, and inferring knowledge from knowledge. As knowledge is knowledge that a proposition is true, these operations on knowledge can also be conceived as operations on the truth of propositions. Logic provides a uniform representational framework for handling truth relevant operations of this kind [Pavelin 1988]. Logic uses basic components like connectives, quantifiers, variables, predicates and constants to build up propositional units which are true or false. It imposes this logical form on propositions to determine which truth inheriting patterns of inference a proposition may legitimately figure in. This representational framework lends itself well to the analysis and synthesis of knowledge and to the extraction of knowledge from knowledge, because it anatomises knowledge into the truth relevant components which determine its referential meaning and govern what knowledge can be deduced from it. Functional abstractions can be used to augment this representational framework to add the ability to cope with abstract datatypes [Xu & Warren 1988] and to use set related classification concepts with built-in inheritance properties [Ait-Kaci & Nasr 1986]. Frame based approaches to knowledge representation [Minsky 1981] can also be reconstructed without loss within a first order logic representational approach [Hayes 1979].

1.2. Declarative Programming

Using a programming scheme based on logic has the advantage that it requires that problems for computation be represented in a declarative fashion - i.e. in terms of propositions which declare the truth about the problem situation. This approach introduces more scrutability into programming, making computer programs wear their meanings more openly rather than in some esoteric formalism with an obscure procedural significance. Declarative programming languages avoid software engineering and performance limitations of other programming languages which have been modelled on the Von Neumann machine [Backus 1978], by evading the operational preoccupation of imperative programming languages with complex state
transitions, and by reducing the potential for bottlenecks in communications between the memory and the CPU. Formulating programs and data in a declarative fashion based on logic should also facilitate natural language processing by exploiting the combination of logic’s inference oriented approach to representing language and logic programming’s model for computing such inferences. Logic programming can be used to express context free rules very naturally using Definite Clause Grammars [Pereira & Warren 1980] and more recent developments have given it full context-sensitive and transformational expressive power and established its aptness for incorporating formal linguistic knowledge into computational linguistic applications [Dahl 1988].

1.3. Automated Problem Solving

The declarative style of logic programming languages helps to realise an aspiration of artificial intelligence research of handing over some of the task of working out how to solve problems to the computer to perform. However, fifth generation computer systems aim at supporting the processing of knowledge and not at embodying artificial intelligence. Their problem solving capabilities are useful, but are hardly intelligent. Intelligent problem solving capabilities must await a future generation of computers which will have mastered as yet unsolved problems in handling knowledge.

Declarative programming is not concerned with describing what must be done to solve a problem. Declarative programming is concerned with describing relevant features of the situation so that the problem to be solved can be posed as a query to the computer. The computer then uses the description and an evaluation strategy to work out how to solve the problem. Logic programming implementations do this by using theorem proving as their basic computation policy. How well they can do this depends upon the combination of the expressive power of their declarative formalism and the sophistication of their evaluation strategy.

The recent emergence of the constraint logic programming scheme [Jaffar & Lassez 1987] has defined a class of rule-based constraint programming languages, which is very significantly increasing the expressive power and problem solving capabilities of logic programming languages. The scheme subsumes logic programming languages as particular cases. Each instance of the scheme realises a declarative programming language over a domain of discourse. Prolog is one instance of the scheme, where the domain of discourse is the Herbrand universe and the constraint solved is unification. The constraint logic programming scheme enables local properties of complex problems to be formulated directly as constraints over an intended
domain of discourse and put together by rules in a logic programming fashion. Thus techniques of arithmetic constraint solving can be combined with resolution to realise declarative programming languages for computing problems stated in terms of arithmetic relationships. Constraint logic programming systems with this kind of capability include CAL [Sakai & Aiba 1988], CHIP [Dinebas et al 1988], CLP(R) [Jaffar & Michalov 1987, Jaffar & Michalov 1987], Prolog-III [Colmerauer 1987], and Trilogy [Voila 1988]. Constraint logic programming looks likely to continue to expand in a major way the scope for handling relationships declaratively in logic programming. It will enable logic programming theorem provers in typed declarative languages to be enhanced with multiple domain specific constraint solvers.

1.4. Highly Parallel Execution

Resolution based logic programming contains extensive scope for parallel execution. This makes it rather promising as a way of articulating problem solving so that very high performance execution becomes possible. The extraction of parallelism can be done largely automatically, which helps remove some of the burden of programming from the user. By contrast the state transition orientation of traditional imperative computer languages induces programmers to formulate their programs in ways which heavily restrict the overall scope for extracting parallelism from them. By shifting computational paradigms to logic programming, much greater opportunity should be revealed for speeding up execution through multi-processing. Once appropriate multi-processing computational strategies can be identified and an implementation technology developed, special purpose hardware based on VLSI technology can be designed to support the multi-processing directly and secure very high performance execution. A crucial limiting factor to attainable performance will be keeping the communication overheads of multi-processing to within sustainable limits.

1.5. Formal Methods

The costs of software development are rising steeply year by year. Annual U.S. software costs are growing at 12 per cent per annum. They were roughly 70 billion dollars in 1985 and will be 125 billion dollars in 1990. Comparable world software costs are estimated at around 140 billion dollars in 1985 and over 250 billion dollars in 1990 [Boehm & Papaccio 1988]. These huge and rapidly increasing costs make increasingly important disciplines like formal methods, which promise to contribute to engineering software more
economically. Proving logic programs correct relative to specifications and proving their computability should be readily feasible, because logic programming is founded in the formally well understood expressive medium of first order classical logic. Logic programming also promises to close the representational gap between specifications and implementations by enabling both to be written in logic, bringing the goal of programs as executable specifications closer to reality [Clark & Tamlund 1977]. In addition, the foundation of logic programming in logic promises to enhance the scope for the use of methods for synthesising programs out of other programs, and for transforming programs automatically in ways which are provably sound [Hogger 1984 chapters 4 and 5].

Enhancements to methods of program synthesis would bolster software writing productivity by making it easier to reuse existing working programs. It would help avoid the endless reinvention and debugging of common types of subprogram for each new application and aid the building up of libraries of programs for faster, more reliable software engineering. Automatic and correct program transformation is also important for removing more of the burdens of software engineering from programmers. For example it enables the benefits of modularity to be attained by allowing a programmer to develop separately the inference engine and the knowledge base of an expert system. A provably correct program transformation technique like partial evaluation can then be used to specialise the inference engine for that knowledge base. The result is a rather faster executing expert system program [Takeuchi & Furukawa 1985].

1.6. Meta-Programming

Logic programming is also very suitable for meta-programming [Sterling & Beer 1989]. Meta-interpreters of logic programming languages are relatively easy to write, and meta-programming provides the basis for a whole variety of ways for handling programs as data, for handling data as programs and for handling the results of programs as data. A flexible meta-programming ability marks off as suitable for artificial intelligence applications languages like LISP and Prolog from traditional third generation programming languages like PASCAL and C. It enables expert and decision support systems to be programmed in logic using flexible inference strategies which can support reasoning guided by heuristics, non-monotonic reasoning, and reasoning under uncertainty. Meta-programming capabilities also allow levels of control in complex programs to be separated out, which enhances their modularity and improves how readily they may be understood. Meta-programming enables more powerful logic programming languages to be built
on top of less expressive ones. Thus the FGCS project uses a hierarchy of logic programming languages from very high level knowledge representation languages like CIL down to the language executed on the highly parallel inference machine KL1-B. Compilation, partial evaluation and program transformation techniques map the higher level languages into the lower level ones [Fuchi & Furukawa 1986].

1.7. Database Query Capability

The success of the relational database model [Codd 1970] has highlighted the merits of schemes of data representation based upon first order logic, and database query languages like QBE [Zloof 1977] have shown how declarative query languages can be used as interfaces to databases articulated by the relational model. Logic programming languages are similar to such query languages in that they are presented to users via declarative query interfaces whose job is to elicit instances under which query propositions can be proved to be true. It is fairly easy to show that Prolog has the expressive power of a relational database system by showing how Prolog can also support all the operations of the relational algebra [Kowalski 1981]. Thus logic programming languages represent a kind of synthesis of a programming language with a database query language. They can also enhance the expressive power of the relational database model by giving it an inferential capability. Deductive databases of this kind [Gallaire et al 1984] are within the existing capabilities of logic programming languages.

1.8. Knowledge Base Programming

Existing experience with the first serious logic programming language Prolog has already demonstrated the merits of a high level logic programming language for rapidly programming expert systems [Clark & McCabe 1982, Hammond & Sergot 1983, Subrahmanyan 1985] and knowledge based systems [Wiederhold 1986]. An established user base has been developed for logic programming languages and techniques. This provides confirmation that logic programming is not just another theoretical innovation from academia which is being ignored by the computer programming community. It is actually feeding into commercial software development practice.
2. Research Context

Research on a fifth generation of computer systems aims at building high performance knowledge processing engines on the basis of the highly parallel execution of logic programs. Logic programming is a declarative computational paradigm founded in a subset of the classical first order predicate calculus, which uses decision procedures based upon resolution to compute answers to queries over programs. It has very considerable potential for supporting highly parallel execution. Practical logic programming languages have been proved to be apt for the development of knowledge based systems, and much active research is going on to expand the expressive power of logic programming languages to enhance this capability. Connections have already been made with other computational paradigms like functional programming [Darlington et al 1985] and a synthesis called equational logic programming has been developed [Jaffar et al 1984]. Connections have also been developed with object oriented programming [Chen & Warren 1988, Conery 1988, Shapiro 1983]. However, outstanding problems in the design and implementation of logic programming languages make it problematic to combine parallel execution, complete support for deduction and a systems programming capability. This combination is important for a major kind of application - the development of knowledge based systems for multiple users on a logic programming system.

Among existing logic programming systems, those based on concurrent logic programming languages like the iPSC implementation of FCP [Shapiro 1987], the Multi-PSI [Ichiyoshi et al 1987] and SPM [Gregory et al 1989], are able to support stream and-parallel execution, which makes them apt for programming systems, while classic logic programming systems for single processor computers like DEC-10 Prolog and Quintus Prolog are able to support a style of execution exhibiting don’t know non-determinism which is effective for complete deduction over knowledge bases. Despite the development of systems for executing logic programming languages in and-parallel (like APEX [Lin & Kumar 1988]), in or-parallel (like Aurora Prolog [Warren 1988]) and in limited or-parallel and and-parallel configurations (like PEPSys [Westphal et al 1987] and LORAP [Biswas et al 1988]), there are as yet no logic programming systems which can effectively program both the systems requirements and the completeness of deduction required for programming knowledge based systems in logic. Various attempts to reconcile stream and-parallelism with don’t know non-determinism have been proposed including Andorra Prolog [Haridi & Brand 1988], Pandora [Bahgat & Gregory 1988] and Parallel Nu-Prolog [Naish 1988]. However, it will be argued in chapter 2 that none is fully suitable to program multi-user knowledge based systems in logic.
This dissertation accepts the framework of assumptions of the fifth generation computer systems research programme and describes the motivation for and design of a logic programming system which reconciles parallel execution, complete support for deduction and a systems programming capability for this type of application. A prototype of the design has been developed in C for a single processor Unix workstation. It demonstrates how the design is capable of being used to support the programming in logic of concurrent scalable multi-user knowledge based systems. The design and prototype are intended as a contribution to research towards the goal of constructing a fifth generation computer system.

3. Research Objectives

The basic objective of this research is to develop the main elements of a practical design for a logic programming system which can handle both the systems and the deductive aspects of multi-user knowledge based systems. Analysis of possibilities translates this into a requirement for a system which is able to support with appropriate grains of concurrency the don’t know non-determinism of search exhibited by Prolog-like logic programming languages, and the stream and-parallelism of the concurrent logic programming or CLP languages, while remaining suitable for parallel execution. A secondary objective of this research is to develop ways for boosting the expressive power of logic programming languages. This is pursued in two principal ways. A method is sought for abstracting from differences between CLP languages so as to realise both a more general CLP language and a more general implementation strategy for supporting several different styles of CLP execution on a single implementation. A method is also sought for supporting escape exception handling in Prolog to support dynamically scoped recovery capabilities for exceptional events.

Assumptions of the design are that it should be portable across a reasonable variety of single and multiple processor machines, that it should be able to run on conventional single processor workstations, that it will extend and build upon existing logic programming implementation technology and that it will be flexible in being able to be interfaced relatively easily with existing logic programming systems. A prototype of the architecture has been developed in C under Unix on a Sun-3 workstation to test out several of the ideas of the design. The prototype has also been developed to interface with a particular software and hardware system termed a Prolog Database Machine [Wong & Williams 1988] which has been developed to enhance the scale of clause databases over which resolution is possible.
4. Research Contributions

This research contributes to the design of logic programming languages, to the development of new programming methods for logic programming languages, and to the development of techniques for the implementation of logic programming languages. Its main contribution is to show how to support multi-user knowledge based systems on a logic programming system capable of parallel execution. It also contributes to the

- design of CLP languages and relationship between GHC and Parlog
- design and implementation of escape exception handling in Prolog
- interfacing backtracking and committed choice resolution engines
- localisation of the GHC run-time suspension test
- design aspects of CLP language implementations

The introduction and chapter 1 attempt to recapitulate the issues involved and the reasoning behind using logic for programming knowledge based systems. Chapter 2 defines a problem in using logic to program knowledge based systems, while chapters 3 to 7 propose an original scheme for solving it. Chapter 8 summarises and draws conclusions.

5. Dissertation Outline

The dissertation consists of an introduction and eight chapters.

Chapter 1 characterises key concepts used in this dissertation’s research including logic, logic programming, parallel execution, knowledge and knowledge based systems and establishes their interrelationships.

Chapter 2 considers the requirements for programming knowledge based systems in logic and how well logic programming languages, which are executable in parallel, are able to program systems and deduction capabilities

Chapter 3 develops a particular approach to reconciling these requirements by coupling committed and trial binding resolution engines together.

Chapter 4 considers suitable CLP languages to be executed on the committed choice resolution engine. The merits of GHC and Parlog are discussed and a strategy is
proposed for reconciling their differences and supporting both in a common CLP language.

Chapter 5 considers aspects of the design of an emulator for supporting this common CLP language.

Chapter 6 considers extensions to the language Prolog to enable it to support escape exception handling and how this might be implemented.

Chapter 7 shows how the unified architecture can be programmed to realise both deduction and systems programming capabilities for knowledge based systems.

Chapter 8 summarises this research and draws conclusions about future developments in fifth generation computer systems research.
Chapter 1

1. Programming Knowledge Based Systems in Logic

1.1. Introduction

This chapter elucidates key concepts used in this dissertation’s research. Logic is characterised, and logic programming is related to it. Key issues involved in executing logic programming languages in parallel are also outlined. The way the two main types of logic programming language relate to the appropriation of a proof procedure for a fragment of first order logic is brought out. The concepts of knowledge and knowledge based systems are also characterised, because the main interest in executing logic programs in parallel is to enable fifth generation machines to process knowledge. The elucidations of this chapter define the issues which set the context for the research problem that this dissertation addresses. The research problem itself is tackled in the following chapter.

1.2. Logic

Logic programming is built upon the twin foundations of the symbolic representations of modern logic and a style of proof procedure called resolution. Modern logic began in 1879 with the publication of the Begriffsschrift [Frege 1879]. Frege’s great advance in Logic was inspired by a major development in Mathematics, Boolean set theory (with influence from De Morgan), just as Aristotle’s earlier development of his Syllogism had been inspired by the development of Euclidean Geometry. In 1910 Principia Mathematica [Whitehead & Russell 1910] rescued Frege’s work from obscurity. It freed Frege’s conceptual notation from unwieldy graphical representations, and enabled the rest of the world to realise the significance of Frege’s new logic.

Since Aristotle first formalised it, logic has tried to formulate the truth inheriting patterns of inference in order to establish a canon of correct reasoning. More recently logic has become conflated with its formal study using mathematical methods. Nevertheless logic is not a purely analytical subject like mathematics. It is a normative analytical discipline, because of its critical role in relation to the practice of reasoning.

The separation between proof theoretic and model theoretic treatments of logic was implicit in Frege’s original work. However, it was not until the 1930s that these methods of treatment were distinguished and
handled formally. This enabled the major characterising properties of the first order predicate calculus to be established. The first order predicate calculus restricts quantification in line with Frege’s original conception to range purely over objects. In the 1930s the first order predicate calculus was established to have the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Nature</th>
<th>Discoverer</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>sound</td>
<td>every provable formula is logically true</td>
<td>Gentzen</td>
<td>1932</td>
</tr>
<tr>
<td>complete</td>
<td>every logically true formula is provable</td>
<td>Godel</td>
<td>1930</td>
</tr>
<tr>
<td>undecidable</td>
<td>no proof procedure exists</td>
<td>Church</td>
<td>1936</td>
</tr>
</tbody>
</table>

Table 1.1 Discoveries of Properties of First Order Logic

Soundness is essential to the point of a formal logic, so it had never really been in doubt. Nevertheless, it was important to establish it formally. Frege’s own set theory had been proved by Russell’s class paradox to be unsound, so soundness is not guaranteed by intuition. Gentzen’s proof theoretic formalisation of first order logic using a sequent calculus shows the impossibility of deriving a contradiction. It demonstrates the soundness of first order logic in a very natural way [Gentzen 1969]. Godel showed first order logic was complete [Godel 1930]. Completeness is also an important property for a formal logic to have. It really represents a criterion of adequacy for a formalisation of logic in being able to prove all that it should be able to prove. Any limits to how complete a formal logic could become, would undermine the aspiration of capturing proof for a logical system in a formal theory. Unfortunately, in 1931 a most remarkable result was proved for suitably expressive formal systems able to support natural number arithmetic [Godel 1931]. It was shown that such systems are incomplete. This incompleteness theorem means that there are formulas in them which are logically true but not provable. In terms of logic Godel’s result means that if quantification is allowed to range over predicates as well as objects, then the predicate calculus would be incomplete. Since Godel had already shown in 1930 that the first order predicate calculus is complete, he had effectively established a limit to the expressiveness of logic - i.e. what could be quantified over, if proof in a formal logic is to retain its essential point of being able to prove each logically true formula. Worse was to come. In 1936 Alonzo Church showed that the first order predicate calculus is undecidable [Church 1936]. This result struck a huge blow at the aspiration of automating the process of proving theorems in logic. Although there are proof procedures for the subset of well formed formulas which are provable, Church showed that
there is no algorithm for the first order predicate calculus that given a well-formed formula can determine in a finite number of steps whether it is provable or not.

Church’s result did not stop research into the development of proof procedures for fragments of first order logic nor did it prevent efforts to develop theorem provers for logic [Chang & Lee 1973], but the lack of a uniform proof procedure for first order logic blocked progress on uniform computational approaches to logic. This situation began to change with Robinson’s work on resolution [Robinson 1965].

1.3. Proof by Refutation

Robinson proposed a refutation oriented style of proof procedure for first order logic based upon a clausal form representation and a procedure of resolution. Clausal form representation is a simple uniform method based on a combination of prenex and conjunctive normal form for representing the formulae of first order logic without the explicit use of quantifiers. A formula in this form looks like

\[ A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m \] where \( n \geq 0 \) and \( m \geq 0 \)

Commas on the left hand side signify disjunctions. Commas on the right hand side signify conjunctions. The symbol \( \leftarrow \) signifies reverse material implication and each \( A_i \) or \( B_j \) is an atomic formula or literal which may have free variables in it. An important limit case is the empty clause where \( n = 0 \) and \( m = 0 \). Of particular interest is the case where \( n = 1 \), the Horn clause form

\[ A \leftarrow B_1, \ldots, B_m \] where \( m \geq 0 \)

To explain resolution requires a few informal definitions. A basic unit at stake is the term. Terms are formed from sets of symbols for functors and variables. Each functor has an arity and functors of zero arity are called constants. A term is a variable, a constant or an n-adic functor of the form \( f(t_1, \ldots, t_n) \) where each \( t_i \) is a term and where \( n \geq 1 \). A literal is a term which is not a variable. An assignment is a set of equations of the form \( \{ v_1 = t_1, \ldots, v_n = t_n \} \) where each \( v_i \) is a distinct variable and each \( t_i \) is a term for \( i \leq n \). A substitution is an assignment where for each \( i \) the variable \( v_i \) does not occur in the term \( t_i \). A term \( t\Phi \) is a term \( t \) with any variable \( v_i \) mentioned in the substitution \( \Phi \) replaced by the term \( t_i \) with which it is equated. A unifier \( \Phi \) of a pair of terms \( t_i \) and \( t_j \) is a substitution \( \Phi \) such that \( t_i\Phi \) is identical to \( t_j\Phi \). A most general unifier is the least set of equations necessary to form a substitution which is a unifier of two terms. It can be shown that if two terms have a unifier, a most general unifier always exists. Resolution is performed by forming
resolvents. The *resolvent* of two clauses C1 and C2,

\[
C1 \quad A \leftarrow B_1, \ldots, B_j, \ldots, B_m \text{ where } m \geq 1 \text{ and } 1 \geq j \geq m
\]

\[
C2 \quad L \leftarrow M_1, \ldots, M_n \text{ where } n \geq 0
\]

where A may be a literal or may mark the absence of a literal, is the clause C3

\[
C3 \quad A \leftarrow B_1, \ldots, B_{j-1}, M_j, \ldots, M_{j+n-1}, \ldots, B_m
\]

C3’s head is the left hand literal of C1, if it had one. C3’s body is composed of the right hand side literals of C1 and C2 less a complementary pair of literals L and B_j, where the pair’s most general unifier substitution has been applied to the clause C3. Thus the resolvent of the two clauses

\[
\text{person}(A) \leftarrow \text{rational}(A), \text{animal}(A)
\]

\[
\text{rational}(\text{socrates}) \leftarrow
\]

is the clause

\[
\text{person}(\text{socrates}) \leftarrow \text{animal}(\text{socrates})
\]

A *refutation* of a set S of clauses is a finite sequence of clauses B_1, \ldots, B_n such that

- each B_i where 1 \leq i \leq n is either in S or is a resolvent of two earlier clauses
- B_n is the empty clause

Resolution is a single inference principle based upon proof by contradiction, which provides a uniform approach to trying to determine automatically the provability of first order logic formulas. The proposition to be proved is negated and both it and the propositions, from which it is to be proved, are expressed in clausal form and form the set S. An attempt is then made to construct a refutation. If a refutation exists, then a contradiction has been derived. This means that the proposition which was negated must be provable unnegated from the rest of S. Robinson’s main result is that a finite set S of clauses is unsatisfiable (has no model) if and only if there is a refutation of S. This shows that refutation is sound and complete. However, it is one thing to know that a formula has a refutation if and only if it is satisfiable (i.e. provable). It is another thing to be able to construct such a refutation. Methods of searching for refutations are called *search principles*. Logic programming employs particular search principles in the attempt to construct a refutation. Because first order logic is undecidable, such search principles are not guaranteed to terminate.
The Horn clause subset of clausal form has proved to be very fruitful for programming the solution of a wide variety of problems [Kowalski 1974, Kowalski 1979]. Resolution search principles for Horn clauses have been discovered which have proven effective methods for constructing refutations. An important such search principle has been SLD resolution. It gave rise to the language Prolog [Roussel 1975]. Prolog combines the expressive power of Horn clause representations with the effectiveness of SLD resolution.

1.4. Logic Programming

Since the development of Prolog based upon SLD resolution, schemes for logic programming have become more complex and more diverse. An excellent survey of many of these schemes can be found in [Clark 1988]. However, Prolog remains the dominant logic programming scheme.

1.4.1. SLD Resolution

Selected Literal Definite clause resolution or SLD resolution can be defined as follows [Lloyd 1984] chapter 2. Suppose the following abbreviations are employed:

\[ \text{Goal}_i = A_1, \ldots, A_m, \ldots, A_n \]

\[ \text{Clause}_i \leftarrow A \leftarrow B_1, \ldots, B_k \]

\[ R \] rule for selecting a literal in a goal

\[ \text{Goal}_{i+1} \] is derived from \[ \text{Goal}_i \] and \[ \text{Clause}_i \] using most general unifier \[ \theta_{i+1} \] via R if

- \[ A_m \] is the literal selected by R
- \[ A_m \theta_{i+1} = A_{\theta_{i+1}} \]
- \[ \text{Goal}_{i+1} \] is \( (A_1, \ldots, A_{m-1}, B_1, \ldots, B_k, A_{m+1}, \ldots, A_n)\theta_{i+1} \)

If P is a program, G is a goal and R is a selection rule, then an SLD derivation of \( P \cup \{ G \} \) is a sequence of goals \( G_0 = G, G_1, \ldots, G_n \), a sequence of variants of clauses in \( P \) \( C_1, \ldots, C_{n+1} \) and a sequence \( \theta_1, \ldots, \theta_{n+1} \) of most general unifiers such that each \( G_{i+1} \) is derived from \( G_i \) and \( C_{i+1} \) using \( \theta_{i+1} \) via R. SLD resolution is sound [Clark 1979] and complete [Apt & van Emden 1982]. SLD resolution can be given a sound declarative semantics based upon an equivalence between a model-theoretic semantics, a fixed point semantics and an operational semantics [van Emden & Kowalski 1976]. SLD resolution can be realised in a logic programming scheme as follows:
Input A logic program P and a goal G.

Output The instance GΦ of G proved from P or failure.

Algorithm initialise resolvent to <- G

\[
\begin{align*}
\text{do } & \{ \\
& \text{pick a goal } A \text{ in resolvent and pick a (fresh) clause } A' \leftarrow B_1, \ldots, B_n \\
& \text{in } P \text{ for } n \geq 0 \text{ where } A \text{ and } A' \text{ are unifiable with substitution } \Phi \\
& \text{if no goal } A \text{ in resolvent is unifiable with any clause head } A' \\
& \text{then exit do loop} \\
& \text{else remove } A \text{ from resolvent and add } B_1, \ldots, B_n \text{ to resolvent} \\
& \text{apply } \Phi \text{ to resolvent} \\
\} \\
\text{while ( resolvent is not empty )} \\
\text{if resolvent is empty} \\
& \text{then output } G\Phi \\
& \text{else output failure}
\end{align*}
\]

Figure 1.1 Basic Logic Programming Scheme

The goal G can succeed or fail with respect to the logic program P. If it succeeds, a substitution is yielded for any variables in G. Having a multiplicity of clauses with which to resolve a goal creates a non-determinacy of choice for a policy of resolution like this. Kowalski distinguishes between two different responses to such non-determinism [Kowalski 1979] p. 114. Those which don’t care and those which don’t know which clause to resolve a goal with. Search principles exhibiting don’t care non-determinism allow for an arbitrary or ad hoc principle of selection. They don’t care what choice is made. On the other hand search principles exhibiting don’t know non-determinism don’t know which to choose, so they consider both choices.

1.4.2. Prolog

Practical use of SLD resolution has to address the possibility that a clause choice might be wrong and prevent the derivation of an empty resolvent, whereas a different choice would not have done so. Prolog solves this problem by employing chronological backtracking to restore the state of affairs where there was last a choice of clauses which are resolvable with the current goal. Instead of the clause which was chosen for resolution, the next suitable clause, which is resolvable with the goal, is selected instead. Prolog’s search
The principle has several important characteristics.

- Depth first search
- Ordered search of clauses from first to last
- Chronological backtracking
- Don’t know non-determinism
- Potentially non-terminating

Prolog’s search strategy is depth first, because it always picks the left-most goal in the resolvent and replaces it by the new goals \( B_1, \ldots, B_n \). It picks clauses, whose heads unify with the picked goal, in their order of appearance in the program from first to last. It uses chronological backtracking, as explained above, as the means of recovering from having picked a wrong clause to resolve with. More intelligent methods are available for determining how far to backtrack [Codognets 1988], but chronological backtracking is easy to implement efficiently. Prolog’s search principle exhibits don’t know non-determinism, because it uses a failure driven search to select the clause for resolution. In this way Prolog solves the problem of not knowing which clause to pick. An obvious alternative to trying each alternative in turn is to try all alternatives simultaneously until a refutation generating selection is found. Prolog allows the use of a control feature called the cut to eliminate consideration of alternatives to the one selected. This introduces a useful measure of don’t care non-determinism to improve the efficiency of execution. Prolog’s search principle is not guaranteed to terminate, because the search for a refutation may recurse indefinitely in the absence of a base case as in the attempt to solve the goal \( \text{<- dead(harry)} \) with respect to the clauses

\[
\text{dead(X) <- corpse(X).} \\
\text{corpse(X) <- dead(X).}
\]

Application of Prolog’s search strategy here results in a fruitless never ending search for a refutation.

Prolog is not suitable, as it was originally conceived, as a target language for fifth generation computer systems, because fifth generation computer systems languages must be executable in parallel. Prolog only performs a sequential search of clause satisfiers, and resolves away conjunctions of literals sequentially from left to right. What is more, this sequentiality is crucially exploited in its efficient implementation [Warren 1983]. However, the success of Prolog as a sequential language on conventional uni-processors showed how resolution based theorem proving could be developed into a practical programming language. It demonstrated how unification could be exploited to support a powerful range of programming techniques. It
also provided an efficient strategy of implementation which looked promising for adapting somehow for use with a version of Prolog which was executable in parallel.

1.4.3. Themes of Parallelism

The primary reason for parallel execution is to speed up processing by doing different parts of it at the same time on multiple processor hardware. A programming language makes parallelism available in computing a problem by enabling the problem to be broken up into multiple sub-problems which can be computed at the same time. It also enables alternative methods for computing the problem to be explored at the same time. There are various desirable qualities for parallelism in the execution of logic programming languages. Parallelism should be

- abundantly made available
- controllable explicitly
- open to implicit extraction
- well exploited

Logic programming languages should make abundant amounts of parallelism available for exploitation. Large knowledge based application tasks contain much that could be executed in parallel, and much parallelism needs to be exposed to harness the computational power of highly parallel machines. Implicit extraction of parallelism is important for relieving programmers of much of the burden of having to plan how to make their programs run in parallel. Both the computational model of a computer language and its programming style should contribute to extracting this parallelism implicitly. Programmers often know in detail about the granularity of parallelism suitable for execution of an algorithm along its critical path. Furthermore, current computational techniques are not always efficient at working it out for themselves. Hence, programmers should be given significant scope for explicit control of parallelism to allow the available parallelism to be well exploited. This will usually require special syntax for annotating programs to induce the requisite control. Well exploited parallelism is not quite the same thing as fully exploited parallelism [Kale 1987b], because full exploitation might be to the detriment of processing throughput. Parallel processing has its own overheads and introduces new and more complex types of processing bottlenecks.

Approaches to executing logic programs in parallel depend crucially on the architectures of the parallel hardware on which they are being supported. Several factors are involved like processor autonomy, memory
architectures, and the connection configuration of the processors. Processor autonomy depends on whether individual processors execute the same instruction stream, the SIMD approach, or whether all processors execute their own instruction streams, the MIMD approach. A general purpose SIMD machine like the CIM-1 connection machine does the first, whereas a MIMD machine like the ALLIANT/FX does the second. The difference is not rigid, as SIMD machines like the CIM-1 can emulate MIMD execution quite readily. Details on programming several parallel machines can be found in [Babb 1988]. MIMD architectures are currently favoured for implementing logic programming languages, but SIMD architectures like the vector parallel supercomputer Hitachi S-820 and the CIM-1 [Nilsson & Tanaka 1988] are also attracting the attention of implementors. Memory architectures depend upon several factors, but the main one is whether shared memory is supported. Parallel execution schemes using global address space like the JAM [Crammond 1988], APEX [Lin & Kumar 1988], and Aurora Prolog [Warren 1988] suit shared memory MIMD machines like the Sequent series whereas purely local address space execution schemes like ORBIT [Yasuhara & Nitadori 1984], and a parallel implementation of FCP [Shapiro 1987] suit non-shared memory MIMD machines like the Intel iPSC series. Currently implementations using global address space are showing the most promise in terms of performance. However, it is generally acknowledged that memory contention in access to shared memory limits the number of processors which can use shared memory, and that the real future for massively parallel computation lies with non-shared memory machines. The FGCS project currently plans to produce a parallel inference machine based on loosely coupled clusters of processors, which attempts to combine the merits of shared memory within a cluster and non-shared memory organisation across clusters [Sato et al 1987].

1.4.4. Sources of Parallelism

The two main sources of parallelism in executing logic programming languages are and-parallelism and or-parallelism. There are also other less important sources of parallelism like parallelism in unification and parallelism in performing search operations over many facts, but these will be ignored here.

1.4.4.1. And-Parallelism

And-parallelism is the parallelism available in resolution when solving goals which have been conjoined or and-ed together. As literal conjunctions are so frequent in definite clause programs, there is clearly a lot of
scope for resolving away conjoined literals in parallel. There are two main forms of and-parallelism

- independent
- dependent

Independent and-parallelism is exhibited, when conjoined goals, having no common variable in their arguments, are executed in parallel. Goals with ground arguments are guaranteed to be independent. However, whether goals with arguments having different variables in them are dependent or not, is determined at runtime by whether these variables are shared or not. Dependent and-parallelism can be

- competitive
- cooperative

Cooperative dependent and-parallelism occurs where two goals communicate via their shared variable. One called the *producer* generates the binding. The other called the *consumer* receives it. Competitive dependent and-parallelism occurs when two conjoined goals compete to instantiate a shared variable. If they both attempt to instantiate it to different values, a binding conflict occurs. So long as unification is atomic, only one of two dependent goals executing in and-parallel, which are attempting to produce incompatible bindings for a shared variable, will succeed. The other goal will be failed and in a computation model exhibiting don’t know non-determinism will be forced to look for an alternative satisfier of the goal. This behaviour is exhibited by Delta-Prolog [Pereira et al 1986]. Ensuring that this covers all satisfaction possibilities has significant overheads. It also raises tricky issues concerning whether to save or discard incompatible solutions from each dependent and-parallel goal.

In order to avoid binding conflicts among literals sharing variables which are executed in parallel, Conery proposed nominating one goal as the producer and the others as the consumers of the binding. Execution of the consumer goals would be delayed until the producer goal had produced a binding for the shared variable [Conery & Kibler 1985, Conery 1986]. The producer-consumer relationship organises the literals into an acyclic directed graph. These data dependency graphs can be constructed at compile time and used at run-time to organise the order of execution of dependent goals. However, all the work cannot be done at compile time since the producer may bind a variable in the goal to a complex term with variables in it. This requires a means for dynamically reconstructing data dependency graphs during execution to establish appropriate producer consumer relationships for these extra variables. There are several variations on
Conery’s model. Some sacrifice parallelism for simplicity and make the data dependency graph static or semi-static [DeGroot 1984, Chang et al 1985] using backtracking algorithms to support don’t know non-determinism [Hermenegildo & Nasr 1986, Chang & Despain 1985]. They have been realised in Warren Abstract Machine based execution schemes [Hermenegildo 1986, Borgwardt 1984]. Others attempt to find more clever means for sustaining the and-parallelism together with an intelligent backtracking scheme [Lin et al 1986, Woo & Choe 1986, Ng & Leung 1988] without losing too much efficiency. Results on a multi-processor implementation are promising [Lin & Kumar 1988], although the completeness of any such logic programming scheme which combines and-parallelism and backtracking is doubtful [Kale 1987b].

1.4.4.2. Stream And-Parallelism

When cooperative dependent and-parallelism is done incrementally, usually by instantiating the variable tail of a list to a partial list consisting of a new element and a new variable tail, stream and-parallelism is exhibited. Stream and-parallel execution allows communication both ways (e.g. via different components of a list element). When coupled with some general dataflow constraints for synchronisation purposes, it enables executing logic programs to become systems of concurrent, communicating processes. This process model makes them particularly suitable for use in systems programming applications [Shapiro 1984, Clark & Gregory 1984, Foster 1988].

The main logic programming languages able to support stream and-parallelism are the concurrent logic programming languages like Concurrent Prolog [Shapiro 1983], GHC [Ueda 1985], and Parlog [Clark & Gregory 1986]. Their model of execution is discussed in more detail later. Both dependent and independent and-parallel execution is enabled for these languages by introducing synchronisation constraints which can be used to determine which goals are the producers and which are the consumers of shared variables. They also allow a restricted form of or-parallelism under the condition that in the search for alternative satisfiers of a goal, variable bindings are not made in the goal which do not contribute to the reduction of the goal.

1.4.4.3. Or-Parallelism

Or-parallelism consists of evaluating at the same time alternative clause choices for resolving with a goal. Actual Prolog programs show quite a significant amount of or-parallelism is available for exploitation [Shen & Warren 1987]. Normally a goal which can be executed in or-parallel will have variables in it, and
rival or-parallel evaluations will compete to instantiate it in conflicting ways. For this reason or-parallelism requires special mechanisms to cope with these alternative bindings for common ancestor variables. Or-parallelism can be employed in two main ways

- competitively
- cooperatively

Competitive or-parallelism is used to speed up the process of finding a single satisfier. On obtaining a single solution, rival parallel efforts examining alternative possibilities for obtaining a solution are abandoned. Cooperative or-parallelism is used to find alternative solutions. Each part of the or-parallel computation cooperates in making a separate contribution to the required set of solutions.

Or-parallel Prologs have been developed which can co-process the branches of a choice-point rather than exploring the left hand branch and then backtracking to the choice-point and exploring the right hand branch. Examples of such systems include the ANLWAM [Butler et al 1986], Aurora Prolog [Warren 1987, Warren 1988], and PEPSys [Westphal et al 1987]. These systems attempt to execute in a coarse grained fashion some choice-points in or-parallel using backtracking to explore the rest. The aim is to take advantage of efficient schemes for executing Prolog sequentially. Other approaches like the Or-parallel Token machine [Haridi & Ciepielewski 1983] execute all alternatives in or-parallel. Purely local memory schemes for or-parallel execution of Prolog on purpose built hardware like the BC-machine are also being pursued [Ali 1988]. The key to any such scheme for or-parallel execution of logic programs is its method of handling multiple binding environments. Many schemes have been proposed [Borgwardt 1984, Crammond 1988] ch.7, Hausman et al 1987, Tinker and Lindstrom 1987, Warren 1987, Yang 1987 ch.6].

### 1.4.4. And/Or-Parallelism

Each sort of parallelism can be used in its own way to support concurrent execution of goals. In particular independent and-parallelism can be realised through or-parallelism [Carlsson et al 1988]. It is also possible to translate some forms of or-parallelism into and-parallelism in the largely and-parallel committed choice languages [Codish & Shapiro 1986]. However, or-parallelism cannot be used to realise fully dependent and-parallel execution and the and-parallel committed choice languages cannot realise full don’t know nondeterminism in resolution as will be seen in the next chapter. Most parallel execution schemes which support either or-parallelism or and-parallelism are able to exploit these equivalences to support some form of
the other. However, not many schemes can combine and-parallel and or-parallel execution together.

An early notable scheme which could support independent and-parallelism and full or-parallelism at the same time was PEPSys [Westphal et al. 1987], although the complexity of the overheads of supporting both at the same time do not make it very efficient. A more recent scheme combines restricted forms of both or-parallelism and and-parallelism [Biswas et al. 1988]. Two other recent schemes combine full or-parallelism and restricted forms of and-parallelism. One based on the AND/OR tree execution model [Conery & Kibler 1985] uses a local memory management scheme [Conery 1987] to execute logic programs on non-shared memory parallel machines. The other based on the REDUCE-OR Process model [Kale 1987a] uses a similar memory management scheme [Kale 1988] to execute on both shared and non-shared memory parallel machines. However, the latter two schemes compromise their efficiency considerably by deliberately avoiding using stacks for memory management. Schemes to be discussed in the next chapter like P-Prolog [Yang 1987] and Andorra Prolog [Yang 1988, Haridi & Brand 1988] can also support restricted combinations of and-parallelism and or-parallelism. However, both these schemes have yet to be proven on a parallel implementation.

An early scheme for full combined and-parallel and or-parallel execution of pure Horn clause programs [Pollard 1981] was highly inefficient and was not complete. More recently an interpreter which combines full and-parallel and full or-parallel execution in a provably complete fashion has been proposed [Raman & Stark 1988]. However, it is acknowledged that a fully distributed implementation of such an interpreter would not be desirable in most circumstances. Full and/or parallelism inherits all the problems with how to exploit or-parallelism effectively. It is difficult to gauge how much or-parallelism to use. Excessive or-parallelism for single solutions execution can result in much competitive work that proves to be fruitless, while too little misses out on speeding up the process of finding a solution. Or-parallelism also creates the difficulty of how to prevent the same goal, which occurs in the same instantiation pattern in several or-parallel branches of a computation, from being recomputed in each branch. And-parallelism can speed up or-parallel search for a single solution by speeding up computation of the adequacy of a particular or-branch. However, it is wasted parallelism if that processing power might be better devoted to exploring other or-parallel branches. Some compromise on exploiting and- and or-parallelism is necessary to secure efficiency in execution. The as yet unsolved question is what compromises are the good ones to make.
1.4.5. CLP Languages

The next most widespread logic programming scheme based upon SLD resolution after Prolog is the approach of the concurrent logic programming languages or CLP languages [Takeuchi & Furukawa 1986] like Concurrent Prolog [Shapiro 1983], Guarded Horn Clauses [Ueda 1985] and Parlog [Clark & Gregory 1986]. These languages developed out of the Relational language [Clark & Gregory 1981] and were influenced by Hoare’s CSP [Hoare 1978]. The CLP languages meet the four broad requirements of parallelism to a significant degree. By being fully and-parallel and allowing a limited form of or-parallelism, they provide abundant scope for parallel execution. Because conjunctions of goals and clause search are executed in parallel by default, they provide considerable scope for implicit extraction of parallelism. The capacity for explicit control of parallelism is the hallmark of these languages, and is proven by their particular suitability for systems programming [Shapiro 1984, Clark & Gregory 1984, Foster 1987]. Several parallel implementations like the multi-PSI's implementation of flat GHC [Ichiyoshi et al 1987], a parallel implementation of FCP [Shapiro 1987], and the JAM’s implementation of Parlog [Crammond 1988] have proven that their parallelism can also be well exploited. CLP languages can be executed either using some global address space as is done for KL1 in PIM clusters [Sato et al 1987] or using purely local address space for a Parlog-like language in a commercial implementation like STRAND88 on an Intel iPSC. A fuller treatment of CLP semantics and the issues they raise is given in chapter 4. However, the general approach to resolution of CLP languages is informally described in terms of two representative languages of this class, namely GHC and Parlog.

CLP languages react differently from Prolog to the possibility that a clause choice might be wrong in the sense that it prevents the derivation of an empty resolvent. Instead of following Prolog in supporting don’t know non-determinism, they expand the space and considerations involved in deciding which clause to select, so as to minimise the risk of getting it wrong. In doing so they embrace don’t care non-determinism instead. This execution policy has become known as committed choice resolution. To this end they use the concept of a guard. The role of the guard is to test whether a goal, which can be unified in the right way with the clause head, should be reduced to the body of the clause.
1.4.5.1. Parlog and GHC Languages

A GHC or Parlog program is a set of relations \(<R_1, \ldots, R_n>\). Each \(R_i\) is made up of guarded Horn clauses of the same name and arity. In Edinburgh Prolog syntax each clause has the form

\[
H : \neg G_1 \text{ AND} \ldots \text{ AND } G_m | B_1 \text{ AND} \ldots \text{ AND } B_n
\]

where \(H, G_1, \ldots, G_m, B_1, \ldots, B_n\) are atomic formulae (unitary Prolog goals) and \(m \geq 1\) and \(n \geq 1\). The clause head \(H\) gives the clause’s relation name and arity. The \(G_i\)s are the guard goals and the \(B_j\)s are the body goals. The commitment operator "|" separates the guarded clause into a passive part consisting of the clause head and guard goals before the "|" operator, and an active part after the "|" operator consisting of the body goals. AND is a meta-symbol signifying a conjunction operator. The primitive true, which always succeeds, fills otherwise empty guards or bodies. The declarative reading of this clause is

\(H\) is true if \(G_1\) and \(\ldots\) and \(G_m\) and \(B_1\) and \(\ldots\) and \(B_n\) are true

AND is a place filler in GHC for the parallel conjunction operator ",", and in Parlog for either the parallel "\," or the sequential conjunction operator "\&". One or more clauses form an ordered relation

\[
C_1 \text{ OR} \ldots \text{ OR} \ C_n
\]

where each \(C_i\) is a guarded Horn clause, OR is a meta-symbol acting as a place filler for a clause search operator, and the symbol "," terminates the relation. In GHC OR only stands for the parallel search operator "," whereas in Parlog it stands for either the parallel "," or the sequential clause search operator ",;". These operators are of equal precedence and right associative, allowing nesting in Parlog. Parlog clauses for the same relation \(R\) of arity \(n \geq 0\) are preceded by a single mode declaration

\[
\text{mode } R(M_1, \ldots, M_n)
\]

where \(R\) is the clause head’s principal functor and each \(M_i\) is either the input argument symbol ? or the output argument symbol ^. All the head arguments of Parlog clauses for the same relation are classified as input or output by this declaration. GHC clause head arguments are always input arguments. To execute a GHC and Parlog program is to refute a conjunction of goals

\[
G_1 \text{ AND} \ldots \text{ AND } G_n
\]

or resolvent where \(n \geq 1\) by input resolution in a fully parallel manner. Resolution succeeds when all goals have been solved so that the resolvent is empty, or fails when a goal in a resolvent obtained from the
original goals fails.

1.4.5.2. Parlog

Parlog execution involves five major operations [Gregory 1987] p. 49

- input matching
- guard goal evaluation
- commitment
- output argument unification
- body goal evaluation

Each goal \( G_i \) is solved either as a primitive by being satisfied and then removed from the resolvent or as a user defined goal as follows. The goal is matched on all input arguments of each clause head for that relation by determining in parallel whether relevant goal arguments can be unified with input arguments of the head of a fresh copy of the clause without binding goal variables or sharing them with each other. Other clauses which may satisfy the head matching requirements, if the goal is instantiated further, are suspended upon relevant goal variables. They attempt to match their input head arguments with corresponding goal arguments, if and when the variables upon which they are suspended are bound. In parallel with head matching, guard goals for each clause are solved. A clause becomes a candidate, if both the input matching and guard goal evaluation succeed, and a non-candidate if either or both fail. It is suspended if either or both input matching and guard goal evaluation suspend. If one or more clauses become candidates one of them is selected non-deterministically. Parallel clause searches are terminated and the goal commits itself to being reduced to the goals in the selected clause’s body. Upon commitment the output arguments of the selected clause head are unified with their corresponding goal arguments in parallel with commencement of evaluation of the body goals. If all clauses are suspended, the goal itself suspends. If all clauses become non-candidates, the goal fails.

Parlog execution is subject to the following special conditions:

- guard safety
- sequential clause search
- sequential goal conjunction
Parlog execution is only valid if evaluation of a guard goal does not bind any variable in an input argument of the clause head [Gregory 1987] p. 51. Ensuring that guards are safe in this way is intended to be done by compile time analysis or run-time checking. Where Parlog clauses have sequential search restrictions on clauses, then a goal only initiates guard goal satisfaction and head matching with all clauses up to the next unencountered sequential clause search operator in order from left to right. Only where a guard goal or a head match fails in each such clause, are the heads and guards of clauses up to the next unencountered clause search operator evaluated. If Parlog goals are conjoined by the sequential operator "&", then only when the goal or conjunction on the left hand side of the operator has been satisfied, is satisfaction attempted on the right hand side goal or conjunction.

1.4.5.3. GHC

GHC is executed by input resolution using parallel clause search and and-parallel goal evaluation like Parlog except that input resolution is subject to the following three rules instead [Ueda 1985]:

- synchronisation
- sequencing
- trust

The synchronisation rule states that trying to bind a calling argument variable in the passive part prior to commitment should cause the unification to suspend. It ensures the goal does not have its variables bound or shared by unification with the clause head. It also ensures safety for guard evaluation, namely that a guard whose clause does not figure in the reduction of a goal will not instantiate the goal during its evaluation. In Parlog input matching and the valid execution requirement related to guard safety perform a similar role to this rule. The sequencing rule states that the clause body may be executed before commitment, so long as the attempt to bind a passive part variable prior to commitment causes the unification to suspend. It guarantees that satisfaction of the body goals won’t affect head unification or guard goal satisfaction until commitment. In Parlog this effect is achieved by not executing body goals until guard goals have been satisfied, input matching has finished, and the goal has committed to the clause. The trust rule states that when a clause solves its guard goals, it tries to commit by confirming that no other clause for that relation has already committed for that goal, and committing uniquely and indivisibly if that is so. The same commitment rule applies to Parlog execution, and makes the execution policy of each examples of committed
choice resolution. Further details are covered in chapter 4 or in the cited references.

1.4.5.4. CLP Issues

Generally speaking, execution of a CLP language takes as input a CLP program and a CLP conjunction of goals. An output is a substitution under which the conjunction of goals can be solved. The process of resolution is the attempt to derive a refutation of it in parallel. Each goal conjunct is solved in parallel by trying to unify it with each clause head for that goal’s relation subject to the CLP language’s synchronisation constraints while trying to solve that clause’s guard goals. Only one of these attempts can succeed, and then the goal is reduced to that clause’s body goals. This resolution process continues until it deadlocks, fails or succeeds. Important characteristics of the execution of CLP languages are

- binding goal before reduction made illegal
- committed bindings and clause choice
- goal reduction non-determinism
- parallel goal evaluation and clause search
- synchronisation constraints
- potentially non-terminating

CLP languages do not allow goal variables to be bound by execution of a clause before resolution commits itself to reducing the goal to the body of the clause. This might happen either during the process of matching the goal with a clause head or during the process of satisfying the clause’s guard goals. This restriction is aimed at preventing clauses which will not be used to reduce a goal from interfering in the process of resolution by generating unwanted bindings in the goal during their head and guard evaluations. This guard safety requirement is discussed in more detail in chapter 4. Another feature of CLP languages, which is related to their need for guard safety, and which makes them relatively easy to implement efficiently, is that once a variable is bound during execution, it remains bound. Unlike Prolog execution their search principle exhibits don’t care non-determinism, once the head unification and guard evaluation of a clause have been performed. Thus the choice of reducing a goal to the body of a clause is a committed one unlike Prolog. It is also one which allows a non-deterministic choice among rival clause candidates for the job of reducing a goal. The combination of committed choice resolution and committed bindings to variables, makes it possible to do without any kind of mechanism for restoring earlier states of the computation. Search is governed
by head unification and guard evaluation under the guard safety rule, instead of by Prolog’s chronological backtracking. CLP languages lose significant search capability here of the generate and test kind [Seki & Furukawa 1987], but gain in being able to support parallel evaluation of goals and parallel clause search easily. Because earlier states do not need to be restored and because bindings are committed, committed choice resolution reveals a lot of scope for efficient multi-processing of goal evaluation and clause search.

Like Prolog it is possible for recursive CLP clauses to invoke each other endlessly and so prevent the program from terminating. CLP programs can also fail to terminate because the computation deadlocks. Deadlock relates to the use by CLP languages of synchronisation constraints. Execution of a goal can suspend waiting upon variables in it to become further instantiated. These synchronisation constraints enable goals to coordinate their execution on the basis of the data available to them. This is crucial for the ability of CLP languages to be executed in a sensible fashion in parallel. However, it remains possible in some programs that all executable goals can suspend at the same time resulting in deadlock.

Logic programming schemes like those of Prolog and the CLP languages are of interest to fifth generation research, because of their potential for supporting knowledge based systems. The CLP languages are particularly interesting, because they lend themselves fairly easily to efficient multi-processing. The FGCS project itself is concentrating its efforts on developing a parallel machine for the highly parallel execution of a CLP language called flat GHC [Fuchi & Furukawa 1986]. How these logic programming schemes relate to knowledge based systems can be seen by developing the concept of a knowledge based system.

1.5. Knowledge Based Systems

A knowledge based system is a system whose functionality is based upon the use of knowledge as knowledge (i.e. in terms of what the knowledge means and implies). Such a system uses a sub-system or knowledge base to manage and maintain knowledge as a resource for the whole system. The system calls upon that resource as required by transacting with the sub-system, but leaves it to the sub-system to maintain and organise that resource otherwise. Contemporary social institutions, language-using human beings, and some modern computer systems are examples of knowledge based systems. In what follows the argument will only be concerned with knowledge based systems which are pure artifacts, namely those that have been realised on computers by programming. The concept of knowledge itself is worth clarifying, as some odd opinions are entertained about what it is [Wiederhold 1986].
1.5.1. Knowledge

Knowledge is what is known. Knowing has two main forms - knowing how and knowing that [Ryle 1949] p.91 ff. Knowing how is procedural knowledge embodied in a skill or capacity like the ability to speak a language or to walk. Knowing that is the form of propositional knowledge, and its propositional nature means it can be about any subject. It can be general or particular, abstract or concrete, theoretical or practical. The condition of being knowledge does not restrict the meaning or reference of the proposition known apart from being related to the truth. The condition of being knowledge is also independent of the ways of representing knowledge. Knowledge can be represented in many different ways including by natural language, by predicate calculus notation, and by frame based representations. Knowing that can be characterised thus [Harman 1968]

someone knows that p  if and only if  he believes that p & p is true & he is justified in believing p

Propositional representations of knowledge are particularly apt, because they preserve knowledge’s internal relation to the truth. Conceived propositionally, the knowledge in a proposition is just that of the truth of the proposition. Logic supports the propositional way of articulating knowledge, because logic is a calculus of propositions concerned with truth inheriting patterns of inference among propositions. Logic also supports knowledge’s relation to justification, because its primary concern, deduction, propagates justification from premisses to conclusions.

1.5.2. Knowledge Bases

Knowledge bases are the parts of knowledge based systems which acquire, store, organise, normalise, and retrieve knowledge. Knowledge based systems differ from other kinds of computerised systems that process significant amounts of symbolic data expressing knowledge like help systems, online manuals, and electronic mail systems. Only they use the knowledge base to handle that symbolic data as knowledge i.e. with reference to what it means and implies. Knowledge based systems include expert systems [Forsyth 1989], databases [Date 1986] and natural language processing applications [Dahl 1988]. The computerised record keeping system or conventional database is the most important kind of knowledge based system nowadays. Such a system does not just use knowledge, husbanding knowledge is its raison d’etre.
1.5.2.1. Database Systems

Conventional database systems are only a primitive sort of knowledge based system, because they are weak at handling knowledge on the basis of what it means and implies. Most modern conventional database systems like INGRES and DB2 are relational, in the sense that their databases are perceived by the user as collections of time varying, normalised relations of various degrees [Date 1986] 11.4. Important characteristics of contemporary database systems are

- efficient, safe and high capacity data storage
- interactive and programmed access on a concurrent basis
- integrated, application-independent data storage
- segmentation by schemas, normalisation and integrity constraints
- model of stored data also stored

Databases store large amounts of data, e.g. the employee and customer records of a commercial organisation, and they use specialised organisation techniques and hardware to manage their storage efficiently and safely. Efficient management of data is crucial for adequate performance and to constrain costs. Safe management of data is important for ensuring disaster proofing, where data loss might have catastrophic effects, and for ensuring secrecy, where this is needed.

Database systems allow various forms of access to data including interactive access through query languages and embedded access from executing programs like a batch run payroll package. Database accesses are allowed by many users at the same time, which enables users to share access to the same piece of data. Concurrency controls are supported to allow updates of shared data and yet maintain consistent views of shared data for different users. This is important for applications like airline booking systems, whose databases are frequently accessed by many users and undergo continual change.

Integrated data storage avoids redundancy, which obviates having to ensure that different representations of the same piece of information remain consistent. Different ways of seeing the same underlying piece of data are supported, allowing different query and programming languages to access data in a format suitable to that application. All such accesses are independent of the details of how and where the data is stored. Different ways of seeing the data can also encompass which items of information and which parts of each stored item of information are accessible. Different permissions to access different parts and items of information can be granted and withheld from different users.
In a conventional database data is stored according to a data model which regiments it into time independent schemas. Data analysis elicits these schemas by examining dependencies among relations in order to determine normal forms for the representation of data. These schemas are also obliged to observe integrity constraints which ensure the consistency of different tables of data with each other. Most modern database systems aspire to being relational databases, which means conformity to the requirements of the relational model [Codd 1970]. This places constraints on entries in database tables like forbidding duplicate entries, and constraints on values of table rows which are used as unique keys for indexing purposes. Update and insertion constraints can also be defined on relations over attribute domains and ranges of numerical values may be defined. These integrity constraints can be stored and evaluated on demand.

Modern database systems also support means of modelling data like data dictionaries and system catalogues [Date 1986] chapter 7, that store data about the data stored which describes its form and the conditions related to its use - details about tables, indexes, views, access permissions etc.. Such meta-data is stored in tables in the same way as any other data. Its availability for retrieval is important for finding out what the database knows about and for effectively managing that stored data.

## 1.5.2.2. Relational and Deductive Databases

First order logic provides both a good representational framework for data in databases and a powerful deductive apparatus for modelling the answering of queries over databases. Recent advances in database and logic programming theory have given rise to a more powerful kind of database system than the relational database, namely the deductive database. The deductive database can be conceived as the addition of the capability of retrieval through deductive laws to the retrieval capabilities of a relational database. It can also be conceived from a formal point of view as the realisation of a first order logical theory with axioms representing deductive laws, integrity constraints and deductive rules and facts [Gallaire 1983, Gallaire et al 1984]. Deductive databases handle these elements in a uniform declarative fashion. The difference between a conventional relational database and a deductive database, can be characterised as the difference between an interpretation of a first order logic theory and the first order logic theory itself [Nicolas & Gallaire 1978], i.e. the difference between model theoretic and proof theoretic views of a first order logic theory. These views coincide if assumptions normally used in relational databases such as
• closed world assumption
• domain closure assumption
• unique name assumption

are formulated as axioms in a deductive database [Reiter 1984, Clark 1978]. Because conventional relational databases are a model of a theory and not the theory itself, they store the base relations of the universe they describe. However, modern relational databases are usually extended to handle derived relations or views which are defined by relational algebra operations over other relations [Date 1986] chapter 8. These derived relations appear to a user rather like base relations which are physically stored in the database. They can be selected, joined and projected like any other relation. However, they are defined as a selection over other relations and have no independent existence. Views enable relations in databases to go significantly beyond model theory. However, views are procedural characterisations of how to derive virtual relations from other relations. They are not treated in the same way as database items like each instance of a base relation. In a modern relational database views are usually subjected to restrictions which do not apply to their equivalents in deductive databases. Views are not allowed to be defined recursively. Instances of derived relations are also not allowed to consist of mixtures of real tuples and tuples derived from executing views or mixtures of tuples derived from executing several views. Views are also significantly restricted in how they can be updated. Furthermore in relational database query languages like SQL the definitions of views cannot even be accessed so that their meanings can be grasped.

Deductive databases rely upon inferential capabilities to draw out the implications of their axioms. They are currently of very significant interest, because the development of logic programming implementation technology and theory suggests that powerful deductive databases can be supported directly or as programs on top of logic programming implementations. Already certain kinds of deductive databases can be supported by the capabilities of existing logic programming implementations [Lloyd & Topor 1986], and moderate extensions of existing logic programming implementations can be used to support definite clause deductive databases [Vieille 1987, Tamaki & Sato 1986, Kemp & Topor 1988]. Indeed certain non-mainstream research programmes into logic programming like the LDL effort at MCC [Zaniolo 1988] are pursuing the ability to support complete set-based querying in a deductive database fashion directly from a logic programming language. At the moment only Horn clause programming languages can be effectively implemented and so deductive databases built on top of them restrict their expressive power to definite information in order to benefit from that efficiency. Deductive databases using full clausal form can express a much
wider set of relationships, particularly indefinite information, but cannot be so efficiently implemented
[Smith & Loveland 1988].

Deductive databases are the logical successors to relational databases. They have several advantages over
relational databases, and a major aspiration of the fifth generation research programme is to speed up the
replacement of relational by deductive databases. The advantages of deductive over relational databases are

- inferential capability
- uniform representational scheme
- greater expressive power
- cleaner declarative semantics
- empowerment by emerging logic programming technology

Although views in relational databases can relate knowledge in the manner of inference, they do not pro-
vide a thorough going inferential capability. Having an inferential capability is the main respect in which
deductive databases go beyond the powers of relational databases. Integrity constraints, queries and deduc-
tive rules and facts are represented in a uniform declarative way in a deductive database, whereas integrity
constraints, views, data and queries are all represented and handled differently in a relational database. The
model theory orientation of relational databases leads to general facts being represented as many separate
instances of a relation, whereas deductive databases can represent general facts much more succinctly and
compactly by representing them directly as universally quantified propositions. In relational databases
views and integrity constraints are restricted in their use (such as being unable to be defined recursively) in
a way that is not applied to their analogues in deductive databases. Thus deductive databases have greater
expressive power. Progress in extending efficient resolution based theorem proving from Horn clause form
to full clausal form promises to enhance this superior expressive power even further [Smith & Loveland
1988] to include the representation and handling of incomplete information. The cleaner separation
between declarative and procedural database concepts for deductive databases also aids the use of theoreti-
cal tools in establishing correctness and consistency properties for databases. Lastly deductive databases
are rather suitable for being supported by the emerging implementation technology and theory of logic pro-
gramming. Progress in the fifth generation computer project and related efforts in other countries i.e. at
MCC, SICS, ECRC, and many universities, promises to improve through logic programming the most thor-
ough going kind of knowledge handling system there is today, the relational database,
1.5.3. Knowledge Base Organisation

The knowledge base of a knowledge based system does not merely embody the axioms and a proof procedure for a first order logic theory. It is also a functional sub-unit of a computer based system, which embodies a control strategy and runs on hardware. When realised in its full form, its major functional components are the knowledge base management system which manages both the use and organisation of the knowledge in the system, and the knowledge base machine, which stores and processes the knowledge.

The knowledge base management system [Mylopoulos 1986] coordinates and manages the activities of query interfaces, query optimisation and satisfaction, inference strategies, explanation facilities, integrity maintenance, knowledge representation methods, security and reliability, indexing and physical storage, and knowledge acquisition and restructuring capabilities. Its precursor on database systems was the database management system. Examples of knowledge based management systems developed at ICOT include KAISER and KAPPA [Itoh 1986] Realising the control strategy of the knowledge base management system and supporting its primary characteristics, like the capacity for concurrent use by multiple users, requires a systems programming approach. A distinctive feature of the FGCS approach is that a host operating system based on a non-logic programming computing paradigm is not called upon to provide concurrency management of these services for database applications tasks. The systems management and the provision of all such services are programmed directly in the logic programming language itself. Thus a logic programming approach to realising a knowledge based system requires both inference programming support for the deductive capabilities of the knowledge base, and systems programming support for the system’s functional capabilities. The real difficulty is to how to achieve this mix.

The knowledge base machine supports the storage of knowledge via a mass storage mechanism and provides a knowledge retrieval mechanism which enables its rapid fetching and combination. Mass storage design is a memory architecture question concerning memory devices, memory layouts, ports, and port networks. Knowledge retrieval design concerns indexing or associative retrieval of knowledge elements stored in the memory architecture and higher order processing of retrieved elements by operations like relational algebra joins or unification. Some approaches to developing a knowledge base machine have separated the facts from the Horn clause rules and attempted to benefit from existing database machine technology by consigning the facts to a relational database machine, the rules to a logic programming system, and the combination to a loose or tight coupling of the two. Examples include ICOT’s DELTA machine which
couples PSIs and relational database machines [Murakami et al 1983] and ECRC’s EDUCE which couples a relational database INGRES and a Prolog system [Bocca 1986]. ICOT refined DELTA to the PHI machine [Yokota et al 1988] to handle recursive queries and access to the external database better. Other approaches have developed techniques for indexing very large numbers of clauses held on mass secondary storage like the superimposed codeword indexing scheme of MU-Prolog [Ramamohanarao & Shepherd 1985], the concatenated codewords scheme of a coupled Prolog-RDBMS system [Shin & Berra 1988] and the balanced and nested grid files of KB-Prolog [Bocca et al 1989].

Recent work at ICOT on knowledge base machines has developed the CHI, PHI and Mu-X systems [Itoh et al 1988]. CHI is a large, shared memory, multi-processing architecture which can execute several sequential Prologs using both dynamic local and static shared clause databases by means of multiple-multiple name spaces. The PHI system is a distributed deductive database system using ICOT’s LAN in order to connect a number of PSI machines. The PSI machines execute an adapted Prolog language and use attached hardware to perform relational operations over function free unit clauses held on disc. Mu-X is a multi-processor backend for PSIs, which executes term-relational queries (extended relational queries that replace equality checking with term unification) over full unit clauses held on large scale multi-port page memory. This work is being integrated with work on the parallel inference machine PIM [Sato et al 1987] by adding a stream oriented primitive rbu/1 to the GHC language being executed on the PIM which will execute term-relational algebra commands over a dedicated relational knowledge base. The design of the PIM and parallel knowledge sub-systems will be refined and realised in VLSI to become ICOT’s prototype of a fifth generation computer.

1.6. Conclusion

This chapter has elucidated the concepts of logic, logic programming, parallel execution of logic programs, knowledge, and knowledge based systems, and related them together. It has also characterised the execution schemes of two prominent types of logic programming languages which will figure in an important way in the argument that follows. The next chapter tackles the problems of designing a logic programming system which will support the programming of both the systems and the deduction capabilities of knowledge based applications while being suitable for execution in parallel.
Chapter 2

2. Resolving Deduction, Parallelism and Systems

2.1. Introduction

This chapter considers how the requirements for programming deduction and systems handling capabilities in logic for knowledge based systems interact with each other. The argument is made that a parallel logic programming system must support both the process interpretation of systems programming and don’t know non-determinism in resolution, in order to realise both the systems and the deduction aspects of knowledge based systems. Support for don’t know non-determinism is needed for the effective handling of deductive capabilities. Support for the process interpretation of systems programming is needed for the effective programming of systems, and support for execution in parallel is needed for adequate performance. However, it has proved hard to combine the ability to support all three in an efficient and expressive way. A number of different ways of reconciling their requirements are reviewed. Their merits are considered in terms of programming a multi-user knowledge base system in logic.

2.2. Programming Proof by Deduction in Logic

A deductive querying capability in a knowledge based system requires the combination of both the following elements:

- knowledge representation scheme
- query evaluation procedure

Deductive querying capabilities can be programmed directly or by meta-programming in the logic programming language. The trade-off is between efficiency and flexibility, although both approaches must meet certain common adequacy requirements. Furthermore, elements of knowledge need to be accessed by knowledge based systems in a variety of modes, which imposes further requirements on the capabilities of logic programming languages. These matters are amplified below to bring out the demands that knowledge based systems make on the characteristics of logic programming implementations. Other demands like the ability to manage knowledge bases across time and space are briefly discussed in chapter seven.
2.2.1. Programming Deduction

There are two main ways to program deductive querying capabilities using logic programming. One represents the knowledge in definite clause form, and uses the built-in theorem prover of the logic programming system directly to determine whether and in what respects the query can be proved over the knowledge base. The other main way of realising a deductive capability for a knowledge based system is to program query evaluation explicitly by a meta interpreter, and to represent the knowledge for that purpose.

Meta-interpretation is much more useful, because its query evaluation procedure can be programmed explicitly to have the desired characteristics [Sterling & Beer 1989]. It enables a choice among a forward chaining, a backward chaining or a mixed inference strategy guided by heuristics, enables non-deductive inference strategies using numerical methods for handling uncertainty to be employed, and allows search spaces to be traversed depth first, breadth first, or in some combined fashion according to requirements, rather than just using the default query evaluation procedure of the underlying logic programming system. However, meta-interpretation runs more than ten times slower than direct execution. Recently techniques like partial evaluation [Takeuchi & Furukawa 1985] have been evolved which can be used to specialise an inference engine, realised as a meta-interpreter, for the knowledge base. They promise to close the performance gap significantly. However, the impure primitives of Prolog [Moss 1986] have created significant difficulties in achieving this fully automatically, and some problems with partial evaluation, like its global control, still remain unsolved [Takeuchi & Fujita 1988].

2.2.2. Evaluation Procedure Requirements

A query evaluation procedure for a knowledge base needs to be

- complete
- efficient
- sound
- terminating

Soundness and efficiency are obvious requirements and create fewer difficulties of realisation than completeness and termination. Although SLD resolution is complete in the sense that each solution belongs to one finite branch of the SLD resolution tree, the query evaluation procedure of a canonical Prolog engine is
neither complete nor terminating. It can be trapped into exploring infinite paths in the SLD resolution tree, and miss finding finite solution paths on other as yet unexplored branches of the tree. A breadth first query evaluation procedure can solve the lack of completeness problem. However, it cannot solve the problem of termination when searching for every one of a finite number of solutions, because the query evaluator does not know when it has found all solutions and should stop. SLD resolution trees can also have an infinite number of solution branches. Furthermore a breadth first query evaluation procedure introduces a new and less tractable difficulty of using very much larger amounts of storage, while being computed, than a depth first query evaluation procedure. Pruning infinitely long branches from the tree using an efficient loop detector [van Gelder 1987] does not solve the problem, because infinitely long branches can have finite side-branches which are solutions, and such pruning would render them inaccessible and make the query evaluation procedure incomplete.

The degree of completeness possessed by the query evaluation procedure of a canonical Prolog system depends crucially on the way don’t know non-determinism in resolution is sustained by its use of a failure driven search for a satisfier of a goal. The Prolog call

\[
\text{\texttt{| - air\_route(berlin, london).}}
\]

over the clauses

\[
\text{\texttt{air\_route(berlin, warsaw).}}
\]
\[
\text{\texttt{air\_route(berlin, paris).}}
\]
\[
\text{\texttt{air\_route(paris, moscow).}}
\]
\[
\text{\texttt{air\_route(paris, london).}}
\]
\[
\text{\texttt{air\_route(A, C) :- air\_route(A, B), air\_route(B, C).}}
\]

succeeds by being able to handle the fact that it doesn't know which flight destination from Berlin to explore in finding an air route from Berlin to London. The attempts to find flights direct to London from Berlin and to find flights from Warsaw somewhere and from Moscow somewhere all fail. These failures prompt further search which enables the route from Berlin to Paris to London to be discovered eventually. Choice points, a variable binding trail, and backtracking in sequential Prolog systems support this don’t know non-determinism in resolution. Or-parallel Prologs combine these facilities with multiple environment binding mechanisms and or-parallel branch points to achieve don’t know non-determinism.
Cuts prune branches of the tree and make execution more efficient, but endanger the completeness of Prolog’s search strategy. If the second rule for `air_route/2` had been

\[
\text{air\_route}(A, C) \leftarrow \text{air\_route}(A, B), !, \text{air\_route}(B, C).
\]

then the top level call would have failed instead of succeeding, because the cut would commit the Prolog engine to trying to find an air route through Warsaw. For this reason knowledge bases represented in Prolog avoid the use of cuts in representing knowledge in the knowledge base. The compulsory cut effect of the commitment mechanism in CLP languages and their associated lack of means to restore previous states of the computation act like the cut in the clause above. This precludes query evaluation in these languages from being complete. This renders them unsuitable for representing knowledge in knowledge bases in a direct fashion. The possibility of programming the query evaluation procedure by means of a CLP meta-interpreter will be discussed later.

When function symbols are allowed in to Horn clause programs, a further cause of non-termination is introduced. The following Prolog program

\[
\text{relation}(X) \leftarrow \text{relation}( f(X) ).
\]

when queried by the goal `relation(A)` results in a non-terminating computation. A simple solution to both difficulties for knowledge bases is to introduce two restrictions

- function-free literal arguments
- stratification of clauses

Making the literal arguments of Horn clause programs function-free, and banning Prolog primitives like `functor/3` which can create function symbols, precludes function symbols from giving rise to problems of non-termination. As the function free restriction is also observed by tuples in relational database, this approach fits a strategy of coupling Prolog to a relational database like EDUCE’s [Bocca 1986].

Stratification or layering addresses the other basic difficulty by precluding SLD resolution trees from having infinitely long branches or infinitely many finite branches. Essentially the idea is to build up the knowledge base in a series of layers in such a way as to prevent recursive clauses from giving rise to infinite computation loops. The base layer of level 0 is the set of clause facts. The next layer of level 1 is the set of clauses, whose body literals are defined only by clauses of level 0. The \(n\)th layer is the set of clauses, among whose body literals at least one is defined by a clause of level \(n - 1\), and none of whose literals are
defined by clauses of level greater than \( n - 1 \). Stratification has been used to define truth in a formal language without recourse to Tarski’s distinction between object-language and meta-language [Kripke 1969], and in logic programming to secure termination in relation to negation by failure [Clark 1978]. These restrictions on admissible forms of representation are quite major restrictions of expressive power especially in ruling out recursive definitions and are probably stronger than is strictly necessary. Weaker restrictions on forms of representation can be framed [Barbuti & Martelli 1986] or the query evaluation procedure can be made more sophisticated to preclude non-terminating computations. However, these restrictions demonstrate that a sound and efficient query evaluation procedure like Prolog’s can be made complete and terminating. This suffices to demonstrate that Prolog incorporates the wherewithal for programming a deductive querying capability for knowledge based systems.

There are several alternative approaches to recursive query processing than the one used by Prolog [Bancilhon & Ramakrishnan 1986]. Recent research has established the existence of top down query evaluation procedures close to Prolog’s strategy like OLD-T resolution with tabulation [Tamaki & Sato 1986], SLD-AL resolution [Vieille 1987] and others [Kemp & Topor 1988] which are complete and terminating in the absence of function symbols. SLD-AL resolution trees have only finitely long branches and the query procedure’s completeness is achieved without placing ad hoc restrictions on recursive Horn clause definitions. Bottom up query evaluation methods can also be used by the implementation to support in a purely logic programming framework complete and terminating query evaluation procedures without restricting the expressiveness of Horn clauses [Zaniolo 1988]. Hybrid strategies of coupling Prolog-like systems to relational database systems are also possible [Miyazaki et al 1986]. However, these methods (a) alter the execution of the underlying logic programming system significantly, or (b) are inefficient when implemented on top of Prolog, or (c) like LDL depend on a quite different implementation strategy from Prolog’s, or (d) like ICOT’s coupled approach, discussed further in chapter eight, compile Horn clause queries into term-relational algebra operations. Their drawback is that while Prolog implementation technology is quite sophisticated, these alternatives cannot yet be implemented with anything comparable in efficiency, or depend on different computational paradigms than logic programming.
2.2.3. Multi-Moded Use of Arguments

Another important feature of a knowledge retrieval capability is its ability to use arguments of relations represented in the knowledge base in a multi-moded fashion. The values of some arguments can be used to match with in order to access the values of related arguments, or can have their values supplied when matching on the values of related arguments. Any argument should be able to be used in either way. Thus the knowledge base clause

\[
\text{divine(venus, goddess, love, roman).}
\]

might be resolved against the goal

\[
| ?- \text{divine(A, B, love, roman).}
\]

or resolved against the goal

\[
| ?- \text{divine(venus, goddess, A, B).}
\]

In the first case the values of the first two arguments are supplied by the resolution process and in the second case they are used in matching. Since unification is a bidirectional operation, logic programming languages based on full unification, like Prolog, have little difficulty supporting multi-moded use of arguments. However, CLP languages, like GHC and Parlog, based upon producer-consumer synchronisation constraints for variables have serious difficulties supporting multi-moded use of arguments. The issue turns on how to represent in GHC a clause like the one above. If it is represented as

\[
\text{divine(venus, goddess, love, roman) :- true | true.}
\]

then although the goal

\[
| ?- \text{divine(venus, goddess, love, roman).}
\]

succeeds, other goals like

\[
| ?- \text{divine(venus, goddess, A, B).}
\]

suspend. The latter goal might be issued to obtain the values of the last two arguments in a clause which matches with the values of the first two arguments. However, the goal would suspend instead waiting for the values of the last two arguments to be supplied by the calling environment. The desired effect for this query can be had by representing the knowledge instead as
divine(venus, goddess, A, B) :- true | A = love, B = roman.

However, if the query is one which matches on the values of the last two arguments

| ?- divine(A, B, love, roman).

it will suspend on this last method of representation. The difficulty cannot be evaded by representing the knowledge by a clause for every possible mode of use. Some are so unspecific in their matching that they could reduce most calls and might get selected by chance in non-deterministic clause selection in place of a more specific relevant clause. The closely analogous approach to input matching and output unification in Parlog means that Parlog displays exactly the same limitations as GHC in this respect. Concurrent Prolog is different. It uses full unification in resolving a goal with a clause head, and so is able to support multi-moded use of facts. Nevertheless the committed choice mechanism of all these languages interferes with multi-moded use of arguments where rules rather than facts are used. Once head unification is performed and the goal is reduced to the body of a clause, there is no going back when it is discovered that satisfying this clause’s body will not yield an answer in the mode required by the query. It remains to be seen whether there is any way of evading these difficulties in CLP languages.

2.3. Programming Systems in Logic

Multi-user interactive systems need to be both responsive to each of their users and to operate in real time. This means that not only must they run on hardware with sufficient raw computer power, but also that they should provide a number of concurrency capabilities including

- dynamic synchronisation of multi-tasking
- fair task scheduling and execution with fine granularity concurrency
- multiple threads of control

Dynamic synchronisation of multi-tasking includes scheduling and control of task execution which can adapt to load, task execution dependencies, task communication and external events. These capabilities are possessed by conventional operating systems and can be invoked by conventional systems programming languages. However, a logic program cannot consign these tasks to conventional operating systems without giving up the attempt to tackle them within its own computational paradigm.
Apart from being able to handle input/output early and asynchronously, a logic programming implementation must fairly distribute opportunities and resources for task execution. This avoids getting uneven and unreasonable responsiveness relative to the load in processing the tasks of each user. Adequate responsiveness also requires a fine granularity of concurrency. It is not enough that each user gets a fair share of available execution power in the long run. That would still be compatible with processing only a few executable tasks at a time. Responsiveness requires that within reasonably small time scales, some progress is being made with executing each user’s tasks. Obviously what is possible ultimately depends upon the rate at which the hardware can process the load, but within the limits which the hardware can handle well, fine grained concurrency is needed to ensure that no task is unduly delayed.

Multiple threads of control are also desirable for handling multiple users and their tasks at the same time. Centralised management in a knowledge based system using only a single thread of control is inefficient in using available parallelism or in exploiting available opportunities for concurrent execution, because it creates a processing bottleneck at the point of allocation of the flow of control. A single thread of control also fails to preserve the natural multiplicity of controlling agencies involved when mapping the structure of several users’ tasks into a scheme for computing them.

### 2.3.1. Models of Systems

If fifth generation computer systems are to be programmed exclusively in logic, they must support these requirements in a logic programming system. This requires a method for expressing and realising systems programming concepts in a resolution framework. Foster classifies the possible models for expressing systems programming concepts, based on various computing paradigms, into three [Foster 1988] chapter 2.

- communicating sequential processes
- dynamic networks of communicating actors
- stream concurrent evaluation of functions

The communicating sequential processes model [Hoare 1978] is based on adding concurrent sequential process creation and synchronous message passing between them to a conventional computational framework. A system of communicating sequential Prologs can fairly easily be realised under Unix on single and multiple processor systems. Sequential Prolog systems can be extended to allow a Unix process executing Prolog to create several others and to communicate directly with them via files or pipes. Both system V and Sun
Unix [Sun OS Manual 1988] also support concurrent execution of processes with shared memory providing monitor and rendezvous mechanisms for process synchronisation, which would allow special send, receive and forking primitives to be used to create synchronously communicating concurrent Prologs. Such schemes would be multi-threaded and would allow dynamic process creation. However, the start up and shut down overheads of creating and terminating each Prolog process would make it efficient to keep the variation in the grain of parallelism fairly static. The operating systems overheads of synchronisation and communication would also make it efficient to keep inter-process communication infrequent. The coarse granularity of the concurrency of such a system, its unwieldiness to change, and bandwidth limits on inter- Prolog communication would make it very hard to map multiple concurrent interacting tasks with a dynamically varying grain of concurrency onto such a set of processes, while retaining both efficient use of processing power and responsiveness and fairness to each user.

Unlike the communicating sequential processes model, the model of systems as dynamic networks of communicating actors [Agha 1986] is an intrinsically concurrent conception. It would seem to fit best into an object oriented computing framework [Goldberg & Robson 1983]. However, some useful analogies exist between communicating actors and communicating nodes in the And/Or tree model of execution of logic programs [Conery 1986]. Stream concurrent evaluation of functions belongs to the functional computation paradigm [Henderson 1982]. However, its conception of modelling input and output between sub-components of a persistent system as infinite data objects or streams, looks as if it would also work well in another declarative computational framework like logic programming. This has suggested the idea of reframing elements of both the actors model and the stream functional model in terms of the controlled deduction computation model of logic programming to express systems programming concepts in it.

A goal can be conceived of as a process, while it is being resolved away. It begins to exist on being created and ceases to exist on being resolved away. Multiple processes can coexist in similar stages of execution if goals are executed concurrently. A process can create one or more other processes dynamically by being reduced to them. A process can also be conceived of as persisting, if it is recursively defined so that it reduces to a copy of itself together possibly with other goals. The resolution tree forms processes into a hierarchy of process dependencies, where parent goals depend for their satisfaction on the satisfaction of one or all of their children goals. Goals conceived of as processes can communicate by passing messages via shared variables. Although goals can execute concurrently in or-parallel as well as in and-parallel, and-
parallelism is needed for message passing. These messages can be passed one way by one goal instantiating
the shared variable to some value and the other goal receiving it. In the following query

| ?- sender(Message), recipient(Message).

the \textit{sender/1} goal can bind \textit{Message} to a value, and this event can be recognised and acted upon by the
\textit{recipient/1} goal. However, it depends upon the \textit{recipient/1} goal being able to delay being executed until the
shared variable is instantiated. Messages can be passed two ways by having respective goals bind respective
members of a pair of variables

| ?- inquirer(Query, Reply), responder(Query, Reply).

or as part of a stream represented by a list of a dialog variable pairs.

| ?- Stream = [ dialog(Query, Reply) | Tail ], transmitter(Stream), receiver(Stream).

The \textit{transmitter/1} goal binds \textit{Query} in a \textit{dialog/2} structure to some value and the \textit{receiver/1} goal responds
by binding \textit{Reply}. The two top level user-defined goals are reduced to themselves with the tail of their for-
mer list argument as their new list argument. The \textit{Stream} is dynamically extended by instantiating the var-
iable \textit{Tail} of the list to another head and variable tail element viz \{ dialog(Q, R) | T \}.

This process interpretation of the execution of a logic program embodies a fine grain of concurrency, uses
stream and-parallelism to communicate between processes and relies upon some kind of synchronisation
mechanism to delay reduction of goals over clauses until message passing variables have been instantiated
by producer goals. It can support dynamic synchronisation of multi-tasking and can sustain multiple threads
of control. However, it does not guarantee any fairness in determining what gets executed when. That
requires a scheduling policy and the means to control and manage that scheduling policy. In what follows it
will be seen that lack of scheduling control is a major sticking point for various parallel logic programming
schemes in being able to support interactive multi-user knowledge based systems adequately.

Goal synchronisation can be done by introducing some general dataflow constraints. A CLP language, like
GHC [Ueda 1985], does this by introducing a rule which makes unification between a goal and a clause
head suspend until the unification is possible without binding or sharing variables in the goal. Although lan-
guages, like Oc and Fleng [Nilsson & Tanaka 1988], mostly use just this suspension rule, it is not very
expressive to use only this condition to synchronise program execution. Introducing special suspension
primitives into the language instead does not suffice to prevent reduction in the presence of general and-
parallelism, because the suspending primitive does not suspend user defined goals executing in and-parallel with it. For a suspending primitive to delay execution of user-defined goals, a computation restriction must be introduced to delay other such goals until the primitive is satisfied. Some sequential operator is needed, as well as a suspending primitive. There is no best way of doing this, although one useful way is to introduce guards. As was seen in the last chapter, guards divide the body of a clause into two sections. Only when the first or guard section is satisfied, can the rest of the clause be executed. Thus a suspension primitive can be placed in the guard, and only when it is satisfied is the body executed. An advantage of guards is that they provide a very general way of framing synchronisation conditions which affect whether and in which way a goal is reduced.

CLP languages use guards, synchronisation constraints and special primitives which can suspend in this way to support just this model of communication between processes in a system. Their and-parallelism allows them to support concurrent execution of goals with facility even on a single processor implementation. Their ability to use this process model to program systems has been well established [Shapiro 1984, Gregory 1987, Foster 1987, Foster 1988]. What makes them particularly suitable for this purpose is their avoidance of support for resolution exhibiting don’t know non-determinism.

2.3.2. Bindings as Signals or Trial Values

The process interpretation of logic programs for expressing systems concepts depends on the idea that the binding of a variable can be used to signal the happening of an event or the passing of a message. However, logic programming implementations, which exhibit don’t know non-determinism, may generate alternative bindings for a variable when executed in parallel and may even withdraw previously made bindings during the course of execution. This means that if a binding is taken as a signal, then it is not finally settled whether the signal is to be taken as sent until the don’t know non-determinism in execution ends. The only alternative is for the programmer to ensure that variable bindings, which act as signals, only get bound once and never get withdrawn once bound. Whether a binding will get undone or not is not always simple to determine. It depends upon complex features of parallel execution, like the scheduling strategy for parallel execution, and what has happened to goals in and-parallel with it elsewhere.

This is rather inconvenient for systems programming purposes, because it interferes with a modular event based approach to managing concurrent tasks. Concurrent task management in systems programming
requires the ability to control and interact with tasks which have arrived at definite states - to put them into further definite states by suspending, resuming or aborting them, and to respond to the results of task execution like success, failure, or even an exceptional or erroneous outcome. If the state of a task depends on some kind of dynamic interplay with the states of other tasks mediated by variable bindings and relationships across the And/Or tree, there is no definite state which any task can be definitely assumed to be in until the dynamic interplay has ceased.

Another way in which don’t know non-determinism interferes with systems programming is with communication. If communication is to be realised as a side-effect of the evaluation of a primitive, it is tricky to interface multi-processed logic programming systems exhibiting don’t know non-determinism to the external world. Unless careful measures are devised, alternative parallel evaluations of the same goal are liable to end up competing with each other in communicating with external agencies at each other’s expense. Multiple demands for input must be arbitrated to ensure that alternative demands for input from the same goal on different or-parallel branches get only one input which is consistent for each branch.

The heart of the difficulty is a fundamental mismatch between a systems programming approach using committed bindings to signal the occurrence of events, and a don’t know deterministic search approach using bindings, multiple alternative bindings, and unbindings to test possible and alternative solutions. A clean separation must be kept between the two uses of variable bindings, if modular and scrutable programming of the systems and deductive aspects of knowledge based systems is to be achieved in logic.

2.4. Knowledge Based Systems in Prolog

It has been seen that Prolog’s ability to support don’t know non-determinism in resolution enables its query evaluation procedure to be complete, sound and terminating for function-free stratified databases, which establishes its ability to support a deductive capability for knowledge based systems. Prolog uses backtracking to find alternative solutions and this unbuilds a canonical Prolog system’s three run-time stacks, enabling the computation space used to find the previous solution to be reclaimed. Also because Prolog uses depth first search, it makes reasonably compact use of memory. It has also been argued that with a minor variation to Prolog’s database management primitives and an escape exception handling extension discussed in chapter 6, Prolog suffices for use in organising a knowledge base. With effective memory recycling, compact use of computation space, a complete search for function-free stratified databases, a
reasonable execution speed and the tools to organise a knowledge base, Prolog is clearly suitable for programming single tasking and single user knowledge based systems.

However, these capabilities are not matched by the limitations which Prolog’s sequential execution strategy places on its ability to handle systems programming tasks. A linear policy of resolution is not suitable for modelling dynamic synchronisation among multiple tasks being processed at the same time. However, coroutining extensions to Prolog have been realised by control annotations in IC-Prolog [Clark & McCabe 1979], by when declarations in NU-Prolog [Thom and Zobel 1986], and by the freeze/2 primitive in Prolog-II [Colmerauer et al 1983] and SICStus Prolog [Carlsson 1987]. These extensions have created a capacity in Prolog for resolving away literals asynchronously in a data driven fashion. They enable these Prologs to offer a limited form of concurrent execution of conjoined goals which can support some of the process model articulated earlier for expressing systems concepts in logic.

In coroutining Prologs the default order of evaluation is still the left to right one for canonical Prolog execution. However, producer consumer relationships can be established for concurrent evaluation using two way communication by exploiting their goal delay mechanism. This can be shown using an adaptation of the example in [Foster 1988] 2.5.4. In NU-Prolog the when declarations associated with a relation delay a goal when called with that variable uninstantiated. Suppose the following query

\[
| ?- A = [X|Y], \ \text{producer}(A), \ \text{consumer}(A).
\]

is made over the following NU-Prolog clauses

\[
?- \ \text{producer}([X|Y]) \ \text{when} \ X.
\]

\[
\text{producer}([X|Y]) \ :- \ \text{test}(X), \ \text{producer}(Y), \ Y = [V|W].
\]

\[
?- \ \text{consumer}(X) \ \text{when} \ X.
\]

\[
\text{consumer}([X|Y]) \ :- \ \text{generate}(X), \ \text{consumer}(Y).
\]

After the top level unification succeeds, the top level producer/1 goal delays on the head of its list argument, and the top level consumer/1 goal is reduced to the body of its clause. The generate/1 goal creates a value for X and thereby wakes up the delayed producer/1 goal. The woken up producer/1 goal is reduced to a test/1 goal which (let’s say) succeeds, a new producer/1 goal which suspends, and a unification which succeeds. This unification enables the consumer/1 goal left over from the previous reduction to be
sufficiently instantiated for the whole cycle to begin again, and the whole process can continue indefinitely building up a list of generated and tested elements on the top level variable A.

This program demonstrates bi-directional communication using streams in the canonical stream and-parallel fashion of the process interpretation of systems programming. However, the program cannot escape its potential for don’t know non-deterministic search. If the test/1 goal fails then both the producer/1 and the consumer/1 goals have to backtrack to generate and test a new value. In this case the backtracking is limited, but in general there is significant potential for complex non-local search. This muddies the tidy model of producer consumer relationships, making it hard to capture systems programming notions in a modular, scrutable fashion. Furthermore the concurrency in this program is not fair, but is only achieved by being always biased towards executing the most recently woken goal first. This means there is no guarantee of giving a fair opportunity for execution to multiple goals which are siblings of each other. For this and further reasons which are argued later, the style of concurrent execution which can be achieved by this means is far less expressive than the concurrency expressible by the CLP languages, and does not suffice to support multi-user knowledge based systems in a properly concurrent fashion.

Or-parallel and and-parallel Prologs have great difficulties doing any better, because they suffer from the basic don’t know non-determinism of the underlying Prolog system. The problem is not one of parallelism but of the difficulty of mixing a systems based conception of bindings as signals with a search based conception of bindings as trial values. Attempts to steer round these difficulties are discussed below.

2.5. Knowledge Based Systems in CLP Languages

Execution models [Takeuchi & Furukawa 1986, Crammond 1986, Gregory 1987] for the sort of fine grained concurrency employed by CLP languages allow a memory management policy for implementations which enables effective and complete recycling of the space used by processes and their arguments, except where complex terms and some shared variables are concerned. The directionality of bindings among such terms precludes simple re-cycling of computation space claimed for evaluating the edges of the resolution tree and requires special support for incremental [Goto et al 1988] or more traditional interrupted execution garbage collection schemes [Crammond 1988] chapter 5. Although computation space recovery in CLP implementations lacks a canonical Prolog’s elegant properties of total run-time computation space recovery on backtracking, it remains manageable and economical.
CLP languages allow literals to be resolved away in parallel, can dynamically synchronise this process effectively using stream and-parallelism, and execute efficiently by pruning exploration of alternative solutions early using a search strategy exhibiting don’t care non-determinism. However, in pruning alternatives early they render themselves unsuitable for finding more than one solution to a query. Thus in making themselves apt at handling the systems programming requirements of multi-user knowledge based systems by sacrificing a complete search strategy, they sacrifice the capability to be sure of finding a solution to a query, if it exists. Furthermore, as has been seen already, the synchronisation constraints of CLP languages, like GHC, preclude multi-moded use of knowledge base clauses. For this reason, and despite all their ability to support the model of systems programming developed earlier, they cannot support the deductive requirements of knowledge based systems directly. Whether they can support the deductive requirements of knowledge based systems by meta-interpretation will be examined in what follows.

2.6. Combined Resolution Policies

Stream and-parallelism enables concurrent systems to be programmed in logic. Resolution exhibiting don’t know non-determinism enables logic programs to support deduction in knowledge based applications. Both are needed for programming concurrent knowledge based systems in logic. The problem is how to square their different requirements efficiently. Proposals for combining the complete search offered by don’t know non-determinism with stream and-parallelism are

- compile away complete search in CLP language
- interpret complete search in CLP language
- compile away stream and-parallelism in Prolog
- devise integrated language capable of supporting both
- couple committed choice and backtracking resolution engines

Each approach has advantages and limitations [Bahgat & Gregory 1988].

2.6.1. Exhaustive Search Interpreter

A complete or exhaustive search interpreter can be written in a reasonably expressive concurrent logic programming language so long as it can perform full unification atomically and reset bound variables in the event of failure. Both Concurrent Prolog and FCP [Shapiro 1987] are able to do this, but other CLP
languages, like GHC and Parlog, only support committed unification and are unable to restore variables bound during unification if the unification as a whole fails. GHC is unable to define full unification in itself, but Clark and Gregory [Clark & Gregory 1985] have proposed getting round this difficulty in Parlog by copying the terms to be unified and then unifying them committedly. If the copies unify committedly, then Clark and Gregory suppose it is safe to unify committedly the originals.

mode unify(\^, \^).
unify(X, Y) :- copy((X, Y), (A, B)) & A = B & X = Y.

However, this method cannot guarantee that between the original moment of copying and the second committed unification, the original arguments will not get further instantiated by other goals running concurrently with unify/2 and sharing variables with unify/2’s arguments. This subsequent instantiation might result in the original arguments to the full unification ceasing to be able to unify.

Fujita [Fujita, 1988] has proposed supporting full unification with variable restoration on failure in FGHC by using frozen representations of all terms and explicit libraries of substitutions. The overhead is plainly enormous, as he acknowledges p. 11. His idea is that by applying an FGHC partial evaluator to the combination of even a very inefficient interpreter of Prolog in FGHC and a Prolog program much of that overhead can be reduced. The result is a general compilation strategy for Prolog programs to FGHC. The approach is interesting as an application of partial evaluation. However, the complex FGHC translation products of even very simple Prolog clauses clearly show that it does not yet provide an efficient method for supporting don’t know non-determinism in a committed choice language.

The price of interpreting exhaustive search in a CLP language is an order of magnitude loss of efficiency in performance and high and not very tractable consumption of memory used in continually building new complex terms to represent goals. Partial evaluation can improve the performance efficiency of meta-interpretation by specialising the clauses relative to the meta-interpreter [Takeuchi & Furukawa 1985]. However, specialisation does not solve the problem of the inefficient use of memory by meta-interpreters. Meta-interpreters in committed choice languages have to evaluate OR-parallel branches of the resolution tree by copying goals at OR-nodes. Each goal copy is a complex term claiming memory in persistent storage space which can only be recovered by garbage collection, because the absence of backtracking precludes the natural stack popping reclamation of memory achieved by Prolog engines on exploring alternative branches. Thus meta-interpreting Prolog in CLP languages is too inefficient a method for sustaining don’t know non-
determinism of multiple solutions search to merit serious attention.

2.6.2. Compiling Away a Complete Search

Ueda has proposed compiling away a complete search over Horn clauses into committed choice resolution using continuations [Ueda 1986b, Ueda 1987]. Similar techniques are used in translating or-parallelism in SCP into and-parallelism in FCP [Codish & Shapiro 1986]. More complex techniques under the same set of assumptions can increase the degree of parallelism exploited [Tamaki 1987, Seki & Furukawa 1987, Okumura & Matsumato 1987]. A language ANDOR-II [Takeuchi et al 1987] has even been defined which has two kinds of relation, AND-relations and OR-relations. AND-relations correspond to FGHC clauses and OR-relations correspond to Prolog clauses. AND and OR-relations are allowed to invoke each other, but similar restrictions to the other approaches are applied, including the rule that input arguments wait until their arguments are ground before being executed. The resulting code is compiled to FGHC.

However, compilation strategies like these have yet to be devised which can handle the full spectrum of requirements of multiple solution search. Existing proposals for automatic transformation require that mode of use information be supplied with the Horn clauses which are to be subjected to a complete search. The mode of use distinguishes whether each argument is going to be used to supply a value or is going to have a value matched with it. Plainly this precludes the kind of multi-moded use of Horn clauses which is often necessary for knowledge based systems applications. Furthermore where logical variables exist which may be written upon by resolution with more than one literal, the so-called multiple writers case, these transformation techniques cannot be applied at all. Similar arguments apply to a recent proposal to combine stream and-parallelism with don’t know non-determinism achieved through backtracking by using a system of precise modes [Samogyi et al 1988].

2.6.3. Prolog and Stream And-Parallelism

A different approach to squaring a complete search with stream and-parallelism is to try to support stream and-parallelism in Prolog. NU-Prolog [Thom and Zobel 1986] is a Prolog extended by co-routining and delayed goal execution features which can support a restricted form of stream and-parallelism, as was shown earlier. The conditions under which a goal may be delayed are given by when declarations which specify when it should suspend and when it may be resumed. These suspension tests do not bind variables
Parallel NU-Prolog is a preprocessor for a parallel extension of NU-Prolog which takes advantage of the binding deterministic subset of NU-Prolog to produce code which can be run in parallel. Binding determinism means that variables shared between goals can only be bound by one goal sharing the variable, and that binding cannot be undone without the whole computation, which created the shared variable, failing. Relations for parallel execution are declared to be eagerly or lazily deterministic and the intended mode of use of each argument of being either input or output is also specified in a Parlog-like way. This information is used to place in the clause body instantiation tests, like \texttt{nonvar/1}, which can suspend, as well as a cut in place of a commitment operator and unifications on which to match input or to output arguments to the goal. When declarations to delay execution of a goal invoking the relation until its arguments are ground are also added to ensure the code is binding deterministic for goals. Guarantees of binding determinism also enable special \texttt{pcall/1} and-forks to be added round goals in compiled code to let the run-time system know the binding deterministic goals which can be executed in and-parallel on another processor. The effect is to transform Parlog-like relations into Prolog which can be co-processed in a stream and-parallel-like fashion.

Thus the Parlog code for unfair non-deterministic merge

\begin{verbatim}
mode merge(? , ?, ?).
merge([D|E], B, [D|F]) :- merge(E, B, F).
merge(B, [D|E], [D|F]) :- merge(E, B, F).
merge([], B, B).
merge(B, [], B).
\end{verbatim}

becomes in Parallel NU-Prolog

\begin{verbatim}
?- eagerDet merge(i, i, o).
merge([], A, A) :- !.
merge(A, [], A) :- !.
\end{verbatim}

which is translated into NU-Prolog

\begin{verbatim}
?- merge(A, B, C) when A or B.
merge(A, B, C) :- nonvar(A), A = D, E, !, C = D, F, merge(E, B, F).
merge(B, A, C) :- nonvar(A), A = D, E, !, C = D, F, merge(E, B, F).
merge(A, B, C) :- nonvar(A), A = [], !, B = C.
merge(B, A, C) :- nonvar(A), A = [], !, B = C.

Input matching is performed by the \textit{when} declaration and the information in each clause's "guard" (the part before the cut) enables NU-Prolog's compiler to determine with which clause to reduce the goal.

In parallel NU-Prolog don't know non-deterministic code can call binding deterministic code, but the two cannot execute in and-parallel. However, the deterministic code can spawn delayed goals which persist afterwards. Binding deterministic code can call non-deterministic code, so long as the call does not share variables with other deterministic calls executing in and-parallel. Don't know deterministic code can also be called by binding deterministic code before cuts so long as inputs to it are ground and outputs to the call are performed after the cut. These combinations allow layered mutual invocations of both kinds of code subject to the stipulated execution restrictions. This enables some quite sophisticated admixtures of a complete search with stream and-parallelism to be achieved. However, the restricted form of stream and-parallelism supported by NU-Prolog is a much less expressive form of stream and-parallelism as compared with CLP languages. Care would also have to be taken to ensure that the constraints on parallel execution did not result in unfair distribution of execution opportunities on a multi-user knowledge based system i.e. lumpy concurrency. In Parallel NU-Prolog

- clause search is always sequential
- guard (pre-cut) goals are always executed and-sequentially
- input matches are always sequential and not concurrent


delayed between shared variables until goal variables are ground
delayed until goal variables in user-defined guard (pre-cut) goals are ground

Guard goals are executed in order from left to right. This is necessary to perform various goal delay tests before input matching unifications. This is in contrast to CLP languages where no and-sequential restrictions on executing conjunctions of goals need exist. The sequencing of execution of guard goals also means that input matching on head arguments is sequential and not concurrent. This follows the practice of most Parlog implementations, although Parlog was originally intended to be executed using concurrent input matching. This means that, where a goal’s first argument is uninstantiated but subsequent arguments are
instantiated, it is possible for a goal to suspend upon the first argument input match even though a subsequent argument input match cannot succeed. When sequential head matching is combined with sequential search, it can result in a goal, which should succeed on a later clause, being suspended on a head match on an earlier clause which it cannot see should fail. Input matching on shared head argument variables as on the arguments of the clause

\[
\text{member}(A, [A\,|\_]).
\]

depends in Parallel NU-Prolog upon delaying execution of the goal until the first argument and the head of the list of the second argument of the goal are ground, and then testing whether they are the same. This precludes using the clause to reduce a goal like

\[
\text{?- member}([2|A]-A, [ [2|B]-B, [2,3|C]-C, [2,3,5|D]-D ]).
\]

using partially instantiated data structures like difference lists. Furthermore where Parallel NU-Prolog’s analogue of deep guard evaluation for CLP languages is employed, which allows calls to non-deterministic code, the stricter delaying condition, that variables used in user defined guard goals are ground, is used. This again heavily limits use of partially instantiated data structures.

For NU-Prolog to support a truly concurrent multi-user knowledge based system on a Prolog system, it needs to be able to schedule in a fair fashion. However, Parallel NU-Prolog lacks any functionality for striking a fair balance among the various demands for multi-tasking that multiple users make when interfaced to a Parallel NU-Prolog system. Lacking even the means for balanced scheduling, of course, precludes any sort of programmable control over scheduling of multi-tasking.

NU-Prolog is an interesting attempt to support a form of stream and-parallelism in a simple and-parallel fashion on a multi-processor. It can support both a complete search and a kind of concurrency suitable for programming systems. However, its ability to express stream and-parallelism is significantly limited compared with the CLP languages. Or-parallelism is not supported at all, and scheduling over processors is highly constrained by the rules for combining the execution of don’t know non-deterministic code with binding deterministic code. Thus fair load balancing relative to each user’s demands is not achieved, which limits the prospects for sustaining fair concurrent execution to multiple users on a concurrent knowledge based system programmed in NU-Prolog.
2.7. Integrated Language Approach

Besides NU-Prolog at least three languages, P-Prolog, Andorra, and Pandora, have been proposed which aim to support stream and-parallelism and don’t know non-determinism in an integrated fashion. Each adopts a different linguistic and implementational approach to reconciling these properties.

2.7.1. P-Prolog

P-Prolog [Yang 1987] uses a different synchronisation mechanism from the instantiation sensitive suspension mechanisms of Concurrent Prolog, GHC and Parlog. Two kinds of clauses are distinguished, exclusive and non-exclusive clauses. Exclusive clauses provide the synchronisation mechanism. In order for one to reduce a goal, the clauses have to obey the sufficient guards principle that only one clause’s head unification and guard evaluation is able to succeed. Until this happens, the goal is suspended waiting upon being further instantiated in order to see whether the sufficient guards condition has become satisfied. P-Prolog uses full head unification in Prolog’s fashion when the sufficient guards condition is satisfied, rather than input matching, which allows it Prolog’s economy of expression as compared with languages like GHC and Parlog. Non-exclusive clauses provide the scope for don’t know non-determinism. They are evaluated in parallel and provide alternative solutions for the goal. P-Prolog also has a special guard primitive other enabling it to force committed choice among different sets of clauses. P-Prolog is able to support both stream and-parallelism using this committed choice mechanism, and to support don’t know non-determinism using non-exclusive clauses.

P-Prolog can be implemented by an Or-tree execution model where each node is represented by a list of and-parallel goals. Whenever a goal can be reduced by more than one non-exclusive clause, P-Prolog requires a virtual copy of the ancestor node to be made for each non-exclusive clause with the goals in the body of each clause being added to the and-parallel list. A multiple environment memory management strategy called concentrative binding has been proposed for implementing P-Prolog [Yang 1987] chapter 6, although any of a number of multiple environment memory management schemes [Borgwardt 1984, Crammond 1988 ch.7, Hausman et al 1987, Tinker and Lindstrom 1987, Warren 1987] could be used to support the or-parallelism of P-Prolog.

In the execution of P-Prolog all potentially exclusive clauses must have their heads test unified with the goal and their guards solved under the bindings of that test unification to determine that only one can
commit. This must happen before it can be decided that any one clause can be used to reduce the goal. That one head unification and guard solution can succeed cannot be used early as in other CLP languages to reduce the goal immediately. Furthermore, because the goal cannot be instantiated before the sufficient guards condition has been satisfied, the evaluation of head unifications and guards for each clause must be conducted in separate binding environments (at least for deep guards). Only when the sufficient guards principle has been satisfied can the local environment be published to the calling goal’s environment. Avoiding the high overhead of maintaining separate binding environments was the main reason why Concurrent Prolog was abandoned in favour of Flat Concurrent Prolog [Shapiro 1986].

Furthermore, where the sufficient guards condition is not met, the goal must be suspended upon all goal variables. Upon any one of these becoming instantiated or shared, the clauses, whose heads can unify with the goal and whose guards have not failed, must be tested again to see whether the sufficient guards condition has yet been met. With deep guards this means recomputing or at least restarting suspended guard computations for each active clause. Plainly restarting the whole sufficient guards evaluation for each new variable binding in the goal is poorly demand driven. Compilation techniques can help determine in advance for certain kinds of guards, simple flat guards, more discriminating and demand driven ways of detecting which variables to suspend the goal on. However, in general with user-defined guards the computation requirements of the sufficient guards condition are high and make P-Prolog an interesting but not very efficient way of combining don’t know non-determinism and stream and-parallelism.

2.7.2. Andorra

Reflection on the inefficiencies of unconstrained or-parallelism in P-Prolog led to D.H.D. Warren’s development of the Andorra model which makes don’t know non-determinism lazy. The Andorra model has been incorporated into various versions of Andorra Prolog including basic Andorra Prolog or Andorra-I [Yang 1988], Andorra Prolog with CLP language features [Haridi & Brand 1988] and an extended version of Andorra Prolog. In what follows only the basic version of Andorra Prolog, Andorra-I, is discussed in detail. Andorra-I distinguishes between a deterministic goal, which can only match a single clause, and a non-deterministic goal which can match more than one clause. Goals which can be reduced deterministically are always executed in parallel first. When no more deterministic reduction can be performed, the leftmost non-deterministic goal is selected and forked, making a branch point, each or-branch is executed one
step and then deterministic reduction is begun again if possible. Or-branches are only created when there are no more deterministic goals to reduce. Thus the don’t know non-determinism is lazy. Like P-Prolog, Andorra-I uses the test for determinism as its synchronisation constraint. However, unlike P-Prolog or-P branches are only created when deterministic reduction is no longer possible. The advantages of the lazy don’t know non-determinism of the Andorra model are that every binding made to and-parallel goals, sharing a common variable during the deterministic phase, is unique to its or-branch, and that search spaces are reduced by limiting the creation of choice points. For better control of the phases of execution Andorra-I allows relations to be declared as AND-relations or OR-relations. AND-relations are only executed during deterministic phases of execution, and OR-relations are only executed during non-deterministic phases.

Andorra Prolog’s approach depends for its success upon how good it is at recognising whether a goal is deterministic or not. Compile time analysis along with simple run-time type tests on arguments, are intended to determine whether a goal is deterministic or not. Warren’s Andorra Prolog philosophy is to try to extract full and-parallelism transparently using compile time analysis without either using the CLP languages’ control constructs of guards or using their (implicit) mode declarations or wait annotations to specify input matching requirements on head arguments. Andorra Prolog aims to avoid recourse to compiler annotations, and to use as far as possible compile time examination of head arguments and early use of built-in primitives in the clause body, in order to formulate conditions for deciding whether a goal is deterministic or not. In other words annotationless Andorra Prolog is intended to have an expressive power similar to flat committed choice languages in programming stream and-parallel applications.

The execution of Andorra Prolog aims at traversing the search space of a logic program in the fashion of a Prolog program. Or-branches need not be Prolog-like choice points. They can be or-parallel forks in the computation. However, they are created and explored in a left to right fashion. This is a deliberate design decision. It is part of the philosophy of attempting to extract parallelism transparently. They are also created lazily one at a time. This policy makes it impossible to guarantee to run concurrently in a fair fashion complete searches for two different clients as might be required by a multi-user logic database system. An Andorra-I implementation will only execute all deterministic goals in and-parallel. As soon as both exhaustive searches run out of deterministic goals to execute, Andorra-I will settle down to expanding the search tree and continuing execution for only one of the exhaustive searches. Only when it finishes will Andorra-I continue execution on the other. The required effect might be achieved if the Andorra model could be
applied to independent sub-computations so that multiple sub-computations could be executed concurrently. However the implementation would need to support mechanisms to ensure rough fairness in allocation of execution opportunities to these sub-computations. This style of development does not seem to be part of the direction of development of Andorra Prolog. Naive global synchronisation of the execution phases of Andorra-I would be likely to result in rather inefficient use of multiple processors. Only when the last processor had finished executing the last deterministic goal could the non-deterministic phase begin. In the meantime the other processors would be waiting doing nothing. Warren aims to address this and other issues by developing the Andorra model to allow lazy non-determinism in each sub-computation and to incorporate other features [Warren 1990]. However, this extended Andorra model does not aim to support even rough fairness in concurrent execution of and-parallel goals.

2.7.3. Pandora

Pandora [Bahgat & Gregory 1988] extends Parlog towards Andorra’s execution model. Pandora adds deadlock procedures and a non-deterministic fork primitive to Parlog. A deadlock procedure is a single extra clause for a Parlog relation preceded by a deadlock mode declaration. When a Parlog computation, i.e. a top level or a meta-called computation, deadlocks and at least one of the deadlocked goals is defined by a Parlog relation extended by a deadlock procedure, then an arbitrary one of these deadlocked goals is executed over its deadlock procedure. A non-deterministic fork primitive is a specially delimited n-wise concatenation of conjunctions of Pandora goals. When executed, it causes the computation to split into n or-branches, each branch defined by its respective conjunction in the concatenation. To avoid the inefficiency pitfalls of P-Prolog the ability to combine general or-parallelism with and-parallelism is restricted so that non-deterministic forks are only allowed to appear as the sole body goals of deadlock procedures.

Pandora is like Andorra-I in that it waits until a computation deadlocks before it engages in non-deterministic execution. It differs from Andorra in that it allows this lazy non-deterministic execution in each meta-called sub-computation. Pandora is limited by its Parlog origins, which requires all Pandora relations to be expressed by guarded definite clauses with their compulsory commit and to be given mode declarations. This limits multi-modal use of relations. Pandora also inherits the limitations of the Andorra approach in respect of scheduling biases. The lazy non-determinism of its deadlock relations would not guarantee fair opportunities for execution to concurrent queries being executed over a knowledge based system.
implemented in Pandora.

2.8. Conclusion

Multi-user knowledge based systems must be able to perform complete and terminating searches of their knowledge to sustain an adequate deductive querying capability. Under certain assumptions, it has been argued that sequential and parallel Prologs can do this efficiently, because they are able to exhibit don’t know non-determinism in resolution. Since the CLP languages cannot exhibit don’t know non-determinism, they cannot support an adequate deductive querying capability directly. It was also seen that the CLP languages cannot efficiently support don’t know non-determinism indirectly, even by compilation using continuations, meta-interpretation by copying, or meta-interpretation using libraries of substitutions. This applies whether or not partial evaluation is employed as well.

It has also been seen that logic programming languages must be able to support the systems programming requirements of multi-user knowledge based systems. Some form of this capability can be realised by giving a fine grained process interpretation to the execution of logic programs, so long as individual goals can be delayed on data-flow synchronisation constraints and some form of stream and-parallel execution is supported for communicating among these processes. CLP languages are the most expressive way of doing this, but analogous capabilities can be demonstrated by and-parallel and co-routining Prologs. However, the process interpretation of logic program execution requires variable bindings to be deterministic vehicles for sending messages. This precludes using them with trial values in a don’t know non-deterministic search. Thus the capacity to exhibit don’t know non-determinism violates an important assumption of the process interpretation of logic programs and in practice completely undermines modular, scrutable systems programming.

Furthermore, co-routining and and-parallel Prologs, like NU-Prolog, are incapable of sustaining other important kinds of systems handling capability required by interactive knowledge based systems like fair scheduling of users’ tasks. The same applies to other attempts to reconcile stream and-parallelism with don’t know non-determinism, like Andorra Prolog’s lazy non-determinism and P-Prolog’s use of the exclusive relation. Thus while Prolog-like languages can program the deductive aspects of multi-user knowledge based systems by exploiting don’t know non-determinism, and while the CLP languages can sustain systems programming capabilities using stream and-parallelism, none of several canvassed possibilities looks
plausible for sustaining the combination adequately. However, one kind of approach is left. It involves the coarse grained coupling of CLP computations with Prolog computations.
Chapter 3

3. Coupling Committed and Trial Binding Resolution Engines

3.1. Introduction

The last chapter established difficulties with reconciling a deductive querying and a systems programming capability in a logic programming language. While Prolog was shown to be apt for supporting a deductive querying capability, it was shown to be unsuitable for systems programming. Conversely the use by the CLP languages of committed bindings and general synchronisation constraints enables them to support systems programming applications quite well, but makes them unable to support a deductive querying capability. This suggests supporting the deductive querying parts of a multi-user knowledge based system with computations using trial bindings in Prolog and the systems programming parts with CLP computations using committed bindings. Their interactions can be coordinated using suitable coupling interfaces to realise all the required capabilities together. This chapter discusses the design of suitable coupling interfaces and methods for realising these forms of coupling. Clark and Gregory’s shared memory approach to coupling Prolog and Parlog is discussed [Clark & Gregory 1987], and a non-shared memory approach to coupling Prolog and CLP resolution engines is proposed instead. A variety of synchronous and concurrent interfaces are defined between the resolution engines, and high level methods of realising these interfaces are described.

3.2. Parlog and Prolog United

Clark and Gregory first advocated coupling different kinds of resolution engine in terms of interfacing Parlog with Prolog [Clark & Gregory 1987]. They defined six interfaces. Three interfaces enable a Parlog computation to call a Prolog computation

- eager all solutions predicate \( \text{set(List', Term?, Conj?)} \)
- lazy multiple solutions predicate \( \text{subset(List?, Term?, Conj?)} \)
- single solutions predicate \( \text{prolog_call(Conj?)} \)

\( \text{set/3} \) and \( \text{subset/3} \) are eager and lazy multiple solutions constructors like Prolog’s \( \text{findall/3} \). They deliver solutions instances of \( \text{Term} \) to the goal \( \text{Conj} \) incrementally as elements of \( \text{List} \). \( \text{subset/3} \) has to be supplied
with the next portion of the List of solutions containing a variable for unifying with a solution element before it will be unified with a solution element. On the other hand set/3 has its List argument unified progressively with the next part of the list containing the solution element eagerly. The third interface prolog_call/1 appears as a Parlog goal to the Parlog system which can execute concurrently with other Parlog goals. Its variables can be bound at any time during its execution either by other Parlog goals executing concurrently with it or by the Prolog computation it represents. These new bindings are visible during the Prolog computation to both it and its external environment. Such bindings by the Prolog computation are assumed always to be committed, which backtracking will not rescind.

Clark and Gregory propose adding a primitive data/1 to Prolog which can suspend a Prolog computation until there is further input. It enables the programming of demand driven computation by the Prolog engine on incremental input via the interface. Since the multiple solutions constructors set/3 and subset/3 can be supported fairly easily using prolog_call/1 as Clark and Gregory show, the real functionality underlying all three interfaces is represented by the third one prolog_call/1 alone. Three interfaces are also defined which enable Prolog to call Parlog

- deterministic conjunction prolog-conj :: parlog-conj
- eager non-deterministic conjunction prolog-conj <> parlog-conj
- lazy non-deterministic conjunction prolog-conj << parlog-conj

Each represents co-routining conjunctions. ::/2 spawns the Parlog conjunction immediately on execution and continues executing the Prolog conjunction. The Prolog conjunction may engage in backtracking so long as no bindings passed to the Parlog conjunction are undone. ::/2 succeeds when both conjuncts succeed. When the Prolog conjunct is true, Prolog is just synchronously invok ing Parlog. The second and third interfaces allow failures in the Prolog conjunction to fail and undo bindings to variables shared with the Parlog conjunction. This rolls back the Parlog computation to the point at which the uncommitted binding, which was undone, was made. If the Parlog computation itself fails, the goal which caused the most recent uncommitted binding in the Prolog conjunction is supposed to be failed. <>/2 allows the Prolog conjunction to carry on making uncommitted bindings to variables shared with the Parlog computation eagerly. <</2 allows the Prolog conjunction to make deterministic bindings to shared variables eagerly yet delays making a binding it could undo on backtracking, and only proceeds if and when the Parlog conjunction deadlocks.
Clark and Gregory aim at exploring the range of possible ways of coupling Parlog and Prolog together with a view to stimulating further research into the design, implementation and use of hybrid don’t know and don’t care non-deterministic logic programming systems. The non-deterministic interfaces $\langle >/2$ and $\langle </2$ represent the really radical departure in Clark and Gregory’s paper. They entail extending Parlog to allow bindings in a Parlog computation to be undone and the computation rolled back. Clark and Gregory devote most of their attention to these non-deterministic operators. Of the two only $\langle </2$ affords sufficient scope for control to be of interest. However, the motivation for the research described here is not to explore the full range of possible interfaces between Prolog and CLP computations. It is to find ways to support multi-user knowledge based systems in logic. This kind of application only really concerns calling Prolog from Parlog-like computations. It does not require non-deterministic interfaces from Prolog to Parlog. Other applications like non-deterministically communicating expert systems and parallel constraint solving systems [Bahgat & Gregory 1988] might require admixtures of don’t know non-determinism with stream and-parallelism of that order, but that is not the concern of this research. Clark and Gregory define interesting and useful ways of coupling Prolog and CLP engines. However, the issues involved need to be examined in more detail to tackle issues they do not deal with, to find less complex interfaces than some of the ones they propose, and to face efficiency issues in the ways in which they can be implemented.

3.3. Implementational Differences

Two issues shape the main difference between implementations of Prolog and CLP languages. Whether

- conjoined goals are executed sequentially or concurrently
- prior computation states are restorable or not

They affect whether stacks can record the program state during execution or whether spaghetti heaps have to be used. They also affect whether bindings and changes to the resolution tree need to be recorded to enable prior execution states to be restored. It turns out that the memory management requirements for concurrent execution of and-parallel goals and backtracking work against each other and would not make memory management of an implementation combining both elements very efficient. Single processor implementations show this most clearly, but the point also applies to multi-processor implementations, which benefit to the extent that they can utilise efficient single processor implementation techniques.
On the canonical efficient implementation of Prolog [Warren 1983], the state of the computation is recorded in arguments and state registers and on the local, global and trail stacks by choice points, environments, term representations and pointers. Forward execution involves updating state registers and argument registers, building new complex terms on the global stack, trailing pointers to bound variables on the trail stack, and discarding or building choice points and environments on the local stack. Backtracking is achieved by going back to the previous choice point on the local stack, restoring the argument and state registers from the choice point’s environment, resetting the global, local and trail stack heights to lower values recorded in the choice point, and unsetting all variables whose bindings are recorded on the trail stack above the new lower stack top. All new building of data structures is performed at the tops of stacks. The whole scheme can be stack based and support backtracking at the same time, because only one branch of the resolution tree is being expanded depth first at a time. Concurrent execution of goals would require one of two things. Run-time data areas would have to have to master the problematic feat of being able to expand multidimensionally at points not at their free ends in a tree-like fashion. Alternatively the simple linear recording of a traversal of the resolution tree would have to be abandoned and be replaced by an interleaved tree representation with no simple unpealing from the top behaviour on backtracking.

On a representative uni-processor implementation of a non-flat CLP language like the SPM implementation of Parlog [Gregory et al 1989] the state of the computation is recorded in argument and state registers and on the process stack and heap by process descriptors, argument vectors and term representations. Execution of processes which succeed involves updating state registers and argument registers, building new variables and complex terms on the heap, and building or discarding process descriptors and argument vectors on the process stack. Execution of processes which fail involves discarding process descriptors and argument vectors on the process stack and returning their frames to the free list (and with an incremental garbage collection scheme would involve adding no longer referenced terms on the heap to the free list). The new building of data structures is done by using the free list to find de-allocated chunks of memory buried in the process stack and heap. Only if the free list is empty or does not have big enough chunks of deallocated memory, is memory allocation done at the free tops of these data areas. Both the process stack and the heap have to be spaghetti heaps of interleaved data structures, to allow the AND/OR tree to be expanded at several nodes at the same time in a non-deterministic fashion. For the same reason slots in the process stack and cells in the heap are not deallocated in an order related to their heights in their data areas.
Adding the capability to restore previous states of the computation would involve saving the values recorded in all deallocated process descriptors, argument vectors and heap cells. Even given optimisations which cut down on what needs to be saved, this would consume much more memory than Prolog requires to record its history. Prolog only records or-nodes explicitly by choice points, whereas a CLP computation records both or-nodes and and-nodes explicitly by CLP process descriptors. Clark and Gregory try to argue that the overheads are not so bad for the <</2 interface as follows [Clark & Gregory 1987] p.954.

"All that is necessary is for the Prolog computation to save the state of a Parlog subcomputation P immediately before each tentative communication with P. When the Prolog subcomputation backtracks past this point [of tentative communication], the saved state of P must be restored. Because we save the state of only those Parlog subcomputations which access the instantiated variable, backtracking in the Prolog [computation] will "roll back" only these Parlog subcomputations, not all of them."

However, if Parlog does not support chronological backtracking in the whole computation, deallocated memory in sub-computations which might be rolled back, must not be re-allocated to the continuation of parts of the computation which will not be. Otherwise it will not be free to be restored as it was. Ensuring that this is so, would have major adverse effects on the compact use of memory by Parlog.

Implementations based on stacks like Prolog unbuild the stacks from the top when memory is deallocated, and are much more memory efficient than implementations like Parlog based on spaghetti heaps. Stack based memory cell management automatically recognises when memory cells are no longer needed without the need for garbage collection. The span of free space is readily defined without the need for threading together deallocated memory cells, and free space is located together for unproblematic allocation in variable width chunks without the need for explicit compaction.

CLP and Prolog computations achieve significant efficiency advantages either by not being able to restore previous computation states or by not supporting general concurrent execution of goals. The first advantage is achieved by embracing committed choice clause selection and committed binding behaviour in resolution. It avoids having to preserve state information on the CLP computation’s history to record variable bindings and prior resolution tree states. The second alternative advantage is achieved by embracing sequential resolution. It allows an efficient stack based rather than an expensive spaghetti heap based memory management policy. The compensation for the first restriction is the ability to support concurrent execution of goals and dataflow constraints among them to synchronise their execution. The compensation for the
second restriction is the ability to support don’t know non-deterministic search efficiently. The two approaches secure their relative advantages in contradictory ways. Compromising between them to try to get concurrency and don’t know non-determinism together would involve sacrificing the respective compensatory advantages which make each of them efficient to implement.

3.4. Coupled Resolution Engines

What is wanted is the ability to invoke multiple Prolog computations in a fully concurrent fashion from a CLP computation, while preserving the respective efficiencies of each type of implementation. Each Prolog computation could handle a knowledge processing task over a knowledge base, and the CLP computation could coordinate and manage the interaction of these tasks and handle any other task requiring significant amounts of communication, control or interaction. However, it is also worth being able to invoke multiple CLP computations from a Prolog computation. This would allow interactive AI applications programmed in Prolog to relegate asynchronous interactions with the user interface to a CLP sub-system, leaving the Prolog computation free to cope with the search problems which are its main forte. The combination of both capabilities should enable CLP and Prolog computations to invoke each other in a flexible, concurrent fashion. The following issues affect interfaces between these types of computations.

- concurrency of components
- memory for shared variables
- revocability of shared bindings
- transient or boundary communication
- view of database clauses

Different kinds of interfaces between Prolog and CLP computations can be characterised depending upon the approach taken to each of these kinds of issue.

3.4.1. Concurrency

Concurrency between multiple computations can be sustained in different ways. The coupling interfaces require that multiple CLP computations be able to execute concurrently with multiple Prolog computations. Multiple CLP computations can already co-exist unproblematically on CLP implementations which support a reasonably versatile meta-call. Each separate computation can be executed using a different meta-call.
This allows a fairly fine grain of concurrency.

Different means for supporting concurrency among many Prolog computations are needed, because canonical Prolog implementations do not support concurrent execution of goals, and co-routining Prologs cannot support fair concurrent execution of goals. In chapter two the possibility was mentioned of supporting multi-threaded Prolog execution using Unix’s concurrency to run several different processes executing Prolog at the same time using pipes, files or shared memory to communicate among them. Multi-processor versions of Unix like the Sequent series’s Dynix operating system [Babb 1988] make this strategy viable for multi-processors as well. However, it was seen that such an approach was not efficient for managing a system of closely interacting and communicating tasks. It embodied too coarse a grain of concurrency. It was inflexible to changes in processing grain because of the start up and shut down overheads of creating or terminating a new Unix process for executing Prolog. It also imposed cumbersome and restrictive bandwidth limits on inter-Prolog communication.

However, having access to a CLP resolution engine means that those tasks of a multi-user knowledge based system which may execute at the same time with a widely varying grain of concurrency but do not require don’t know non-deterministic execution, can be executed on the CLP resolution engine. This includes interface management, input/output handling, systems coordination, task scheduling, and other control related jobs. Only tasks like query satisfaction, integrity checking, explanation construction and perhaps query optimisation require don’t know non-deterministic execution. However, only a few of these latter tasks are typically required for each user at a time. They do not require to interact tightly with other tasks. Thus coarse grained concurrency in the execution of sequential Prologs would suffice to process them.

This suggests supporting a multi-user knowledge based system on a coupled CLP-Prolog system by using one Unix process (or several tightly interacting Unix-like processes sharing memory on a multi-processor) to execute the CLP language and one or a few Unix processes per user to execute sequential Prologs. Unix sustains overall concurrency among the multiple sequential Prologs and the sole CLP engine. The CLP resolution engine can process the closely interacting and communicating tasks of the knowledge based system as CLP computations with fine granularity concurrency. The remaining knowledge processing tasks requiring don’t know non-determinism can be processed with low degrees of interaction and coarse granularity concurrency by a few separate sequential Prologs per user. The whole system executes a CLP program on the CLP engine and separate Prolog programs in each Prolog process.
However, sometimes it would be desirable to sustain concurrent execution of multiple Prologs on a more closely coupled basis where the Prologs would share memory. This could be realised under System V Unix using its shared memory facility or realised under Sun Unix OS 4.0 [Sun OS Manual 1988] using lightweight processes sharing the memory of a single Unix process. A conventional Prolog engine might be adapted to run as a multi-threaded Prolog engine with interleaved stack segments for each thread of the computation. A scheduler could switch the engine between threads on a resource or time sliced basis or under explicit control of a programmable scheduler. The engine could organise stack handling of the interleaved memory map directly or the same effect might be achieved indirectly by using a virtual memory organisation and separate trios of virtual run-time stacks for each Prolog computation. The run-time data areas would no longer be true stacks but would be interleaved throughout with inviolate state information for other computations. However, continual range checking of the scopes of each of the three current stack segments for each computation, and periodic pushing or popping of stack segments as the interleaved stacks expanded or contracted, would impose a heavy overhead on memory use and threaten to ruin the efficiency of the basic WAM engine. A better approach would be to divide up the shared memory so that each of these separate Prolog computation threads had its own non-virtual local, global and trail stacks. They would only need to share other data areas like the code area and symbol tables using a concurrency control mechanism such as monitors to handle updates to these areas.

### 3.4.2. Memory for Shared Variables

Clark and Gregory advocate a shared memory approach to storing variables shared between Prolog and Parlog computations [Clark & Gregory 1987] section 4. They allow each of a coupled pair of Parlog and Prolog computations to bind variables in each others heaps. A non-shared memory approach would make each Prolog and CLP computation maintain a separate binding environment and copy bindings across the interfaces both ways. Clark and Gregory’s approach is less desirable for several reasons.

- efficient variable management is interfered with
- garbage collection becomes highly problematic
- loosely coupled multi-processing is made inefficient
- modular development of coupled systems is undermined
Avoiding the creation of dangling pointers in logic programming implementations depends on determining the direction of variable to variable bindings by the respective addresses of variables. Thus in the WAM implementation of Prolog [Warren 1983] variables higher in the stack (local stack) or heap (global stack) may be bound to variables lower in their own stack and stack variables may be bound to heap variables, but not vice versa. These binding directions are enforced during unification by ensuring that higher addresses are always bound to lower addresses, and by ensuring that the heap is placed below the stack. If bindings are allowed across multiple extra heaps, simple tests like this cannot be used to enforce variable binding direction policies. For the same reason garbage collection is made much more difficult, because an intertwined spaghetti heap of variable pointers across multiple heaps must be traversed before dead space can be safely identified. Loosely coupled multi-processing is also desirable for concurrent execution of coupled CLP and Prolog computations, especially to achieve highly parallel execution. Reducing avoidable sharing of memory areas is crucial to sustaining its efficiency. Lastly, it is important for a coupling strategy to be modular so that it can remain up to date. A coupled execution strategy must be able to keep abreast fairly easily of the latest developments in the implementation technology of CLP and Prolog engines. This will be possible only if the coupling strategy only requires minor adaptations to the workings of each engine of a coupled system. Otherwise a coupled system is liable to become a specialised idiosyncrasy which is bypassed by developments. For these reasons all bindings will be made local to their own computation, and changes to shared bindings will be copied across between computations.

3.4.3. Revocability of Shared Bindings

There is a simple way of reconciling the differences between a CLP and a Prolog implementation, without undermining either Prolog’s efficient stack based implementation, or a CLP computation’s lack of need to restore prior states of the computation. It is to insist that Prolog bindings made to variables shared with a CLP computation cannot be revoked once they are communicated.

**Irrevocable Communication** bindings communicated to a CLP computation cannot be undone by the Prolog computation

Violating this restriction causes the whole Prolog computation to fail (or raise an appropriate escape exception as discussed in chapter six). This requirement is more permissive than Clark and Gregory’s requirement of not rescinding bindings which have been made. It allows a Prolog binding to a shared variable to be
revoked so long as it has not been communicated. This restriction is advocated instead, because it fits in with a loose approach to sharing variables using two way copying across the CLP-Prolog divide rather than with Clark and Gregory’s tight approach to sharing variables via common memory. The restriction allows the Irrevocable Communication restriction to be enforced by the interface independently of the internal workings of the CLP and Prolog engines, which is important for modularity.

Binding irrevocability across interfaces would rule out complex interfaces like Clark and Gregory’s non-deterministic invocations of Parlog from Prolog &./2 and &.</2. Three reasons for doing this are to

- keep semantic simplicity in the interface constructs
- preserve the systems programming model of CLP languages
- retain efficiency in the implementations coupled together

Allowing revocable bindings to be passed across the interfaces from Prolog to CLP computations would make the semantics of the interface rather complex. Programs using such interfaces would not be very scrutable, and that runs counter to an objective of FGCS research of enhancing software writing productivity by retaining semantic transparency. The last chapter brought out another difficulty with allowing revocable bindings to variables. They interfere with the role accorded to variables in the process interpretation of systems programming of being the message passing medium for concurrent, communicating processes. Messages, once sent, have happened and cannot be undone. However, if revocable Prolog bindings could be shared with a CLP computation and then rescinded in both, the CLP computation could be subject to such non-determinism. Systems supported on top of it on the basis of the process interpretation of systems programming would be subject to a strange kind of retroactive interference which would rescind messages sent between sub-systems, as if they had never happened. Thirdly, by not allowing Prolog computations to rescind communicated bindings and initiate rollbacks in CLP computations which have consumed and acted upon that binding, the efficiency of existing approaches to implementing CLP resolution engines would be preserved.

3.4.4. Transient Communication

Another issue for coupling CLP and Prolog resolution engines together is whether bindings to shared variables should be made available transiently during the computation or only at the invocation boundaries of a computation. Clark and Gregory have argued for having transient communication from a Prolog
computation to a CLP computation, in order to maintain a reasonable degree of concurrency in the CLP computation. Thus while a Prolog computation generates successive elements of a list, they can be incrementally communicated to the CLP computation and dealt with as they are made available. Corelatively as a CLP computation generates successive elements of a list they can be incrementally given to a Prolog computation. However, incremental communication raises the issue of having to cope with binding revocability discussed earlier. Because of this kind of difficulty, boundary communication is sometimes desirable to prevent premature communication of bindings from Prolog to the CLP computation which the Prolog computation is committed not to revoking. For example in a Parlog computation a conjunction of goals might be given to a Prolog computation to run in parallel with another Parlog goal.

Parlog ?- prolog_call( (generate(A), test(A)) ), consume(A).

Suppose the Prolog computation satisfies generate(A) by binding A and this is communicated early to the Parlog computation before the prolog_call/1 invocation is finished. Suppose further that the Prolog goal test(A) then fails for that binding and initiates backtracking. The binding cannot unproblematically be revoked by resatisfying the goal generate(A). Revoking the binding would threaten violation of the Irreversible Communication restriction. This would force the failure of the whole top level conjunction in Parlog. It would be better for this kind of invocation to delay communicating any Prolog bindings to the Parlog computation until any backtracking in the Prolog computation has finished. Both transient and boundary communication interfaces have merits, so it is better to support both.

### 3.4.5. View of Database

Typical Prolog systems have a single clause database which can be changed by adding or removing clauses. The database state persists between top level queries, so that changes made to the database, while computing one query, permanently affect the database state for subsequent queries. All of the database is visible at any time and during any query computation. This makes reasonable sense for interactive use by a single user. However, where multiple Prolog computations are being invoked concurrently from a CLP computation, these Prolog computations may not want or need to share a common database with a single persisting state, which all can see all of, and which any of them can change. At least two issues are involved.

- sharing of databases
- state persistence of databases
Logic programming systems need to be able to organise knowledge bases spatially and across time. If the database contains many clauses, facilities for partitioning it into manageable sub-sections are useful. Also if separate versions of the database are recorded to support applications like assumption maintenance and temporal reasoning systems, and integrity maintenance during retrievals and updates to a shared database, it is useful to have facilities for managing versions of the database. Classical Prolog systems can handle database partitioning and version control of clauses relatively easily. They support primitives which can flexibly manipulate file sized chunks of clauses, and primitives which make it easy for the purposes of version control to recognise and access clauses, and to copy, rename, and label clauses.

Two Prolog computation threads invoked from a CLP computation may only access two different files of clauses. So loading the clauses into separate databases would make sense to avoid memory access contention on a shared memory multi-processor. It would also avoid the unnecessary overheads of operating any concurrent access regime with multiple Prolog computation threads on a single processor implementation. Where two different Prolog computation threads want to share access to common clauses, whether they want to execute and read them or to update them as well, will determine whether the clauses should be loaded into a shared or private databases. If the clauses are only executed but not updated, it would be wasteful to load them into two separate databases. It would make sense to have a single executable version of the shared clause code accessible to more than one Prolog computation thread. If common clauses are going to be both executed and updated, whether the clauses should be shared or loaded into private databases depends upon the purposes of the each computation thread. The clause changes might be a local matter for the individual Prolog computation thread which are not intended to persist after the Prolog computation thread has finished. Equally the clauses might be intended to be made public and to affect concurrently executing Prolog computation threads. The latter issue raises the question of concurrency controls discussed in chapter six and database persistence discussed below. However, it can be seen already in the coupled CLP-Prolog system that different Prolog computation threads need to have their own code areas as well as being able to share code areas with other Prolog computation threads.

The state of typical Prolog databases persists between top level calls from start to finish. This is convenient for interactive use. A file of clauses can be consulted and then a query can be made over the revised database of clauses. However, conventional Prologs do not preserve the database state between exiting from the system and re-entering it. Preserving the updated state of the database has advantages, and the
capability is useful enough for many commercial Prologs to allow the updated state of a Prolog system to be saved for use as the initial state of a Prolog system. Its major drawback is that state preservation takes time to perform. Persistent Prologs [Colomb 1989] make this facility more sophisticated, so that the state of the database of a Prolog system can be maintained relatively independently of whether clauses have been added to an original state. Thus clauses can be made visible just by opening a persistent database of clauses, rather than by being consulted, compiled or loaded in, as happens to externally held clauses in Prolog-X in the Prolog Database Machine [Massey et al 1989] (see also chapter 7). Persistent Prologs like NU-Prolog [Thom and Zobel 1986] achieve these capabilities by using slower mass storage memory like disc to store clauses. Slower retrieval times to such clauses are compensated for by using clause indexes held in main memory [Ramamohanarao & Shepherd 1985, Shin & Berra 1988, Bocca et al 1989]. Clearly it would be useful if a CLP computation could invoke a Prolog computation and set up a database state which would persist even after the Prolog computation finishes. Subsequent accesses would use the already initialised database setup. It is also important that an invocation of a Prolog computation can avoid having to use an already loaded database state and can use the unloaded initial database state instead. Thus a coupling scheme of CLP computations to Prolog computations should allow Prolog computations to have

- private and shared databases
- persisting inheritable or uninitialised new database states

for adequate flexibility. In order to allow for the simple manipulation of these recommended relationships of Prolog computations to clause databases, the following principle will be adopted.

**One Database per Meta-call** each separate CLP meta-call environment is associated with a separate persisting Prolog database state.

This database state will be created by the first call to a Prolog computation from within that meta-call’s environment. Subsequent to that, each call to a Prolog computation within that meta-call environment will access the same database state. Once created that Prolog database state will persist for as long as the CLP meta-call environment persists. Prolog invocations within different CLP meta-calls will always be to different Prolog database states. In terms of Clark and Gregory’s `prolog_call/1` interface, this principle means that the and-parallel Parlog goals
Parlog ?- prolog_call(smash(A)), prolog_call(grab(A)).

would execute concurrently over the same database state, because they are made within the same meta-call. On the other hand the and-parallel Parlog goals using the Parlog meta-call \textit{call/3}

Parlog ?- call(prolog_call(smash(A)), _, _), call(prolog_call(grab(A)), _, _).

would execute concurrently over two separate Prolog database states. This kind of arrangement is simple to understand and use. It can be realised fairly easily using lightweight processes sharing a code area and symbol table to realise concurrent Prologs sharing the same database, and real Unix processes to realise concurrent Prologs with their own private database states. It is even compatible with a means whereby different Unix processes executing Prolog can share a common set of clauses, such as the mechanism described in chapter seven for a backend database machine for the host Prolog engine used in the prototype CLP-Prolog system, which tested some of these ideas. Further examples to illustrate these different modes of invocation and their significance for concurrent use, shared database access and database state persistence are given in chapter seven.

### 3.5. Computation Interfaces

The various needs for coupled resolution require a variety of coupling interfaces. The following interfaces are proposed as an integrated set of ways for coupling CLP and Prolog systems. These interfaces assume that the environments of the two resolution engines are separate. All communication across interfaces is irrevocable and involves copying bindings across. In this way the versions of the goals on each side remain able to unify, but do not store shared variables in a commonly accessed memory. Each communication is atomic, with no bindings being made if the goal versions cannot unify. If such a unification is impossible, the invoked computation fails with normal goal failure consequences for the invoking environment. Two interfaces allow Prolog to call a CLP computation as follows:

- call to CLP relation
  - communication at boundaries
  - input argument suspension on invocation possible
  - synchronous
call to \textit{clp}(Goal)

- communication transiently both ways (with a restriction)
- input argument suspension on invocation of Goal possible
- asynchronous

Calls using these interfaces are never resatisfiable. Goals in a Prolog computation, which are not defined by a clause relation and are not calls on a primitive, invoke the CLP resolution engine. The invocation is synchronous, and values are passed at the boundaries of invocation without any concurrency between the calling computation and the computation which is called. Input matching suspension on invocation takes place if needed. This interface is essentially a version of the interface proposed by Clark and Gregory’s deterministic meta-conjunction operator \texttt{true :: relation(Arg1, ..., Argn)} where \texttt{relation/n} is a CLP relation. Calls in Prolog to \texttt{clp/1} will invoke the CLP resolution engine asynchronously. \texttt{clp/1} goals in Prolog coroutine with all siblings goals in its clause body, or if invoked at the top level with all other top level goals. To keep the scope of its concurrency simple, \texttt{clp/1} goals are not allowed to appear in goal bodies with a disjunction \texttt{;} or a conditional operator \texttt{<-}. The parent goal of the spawned call to the CLP resolution engine does not exit until the \texttt{clp/1} call has succeeded or failed. When a \texttt{clp/1} goal fails, Prolog backtracks to look for another satisfier of the parent goal. The behaviour of \texttt{clp/1} is similar to Clark and Gregory’s deterministic \texttt{::/2} operator, although no scoped conjunction operator has been introduced.

Argument values to shared variables may be passed transiently during \texttt{clp/1} invocation subject to the following restriction on shared variables

\textbf{No Complex Term CLP Bindings} \texttt{CLP execution may not bind variables shared with a Prolog computation to complex terms during execution}

A CLP computation is only allowed to bind shared variables to simple terms while executing a \texttt{clp/1} call (or its transient analogue \texttt{prolog/1} from CLP to Prolog defined later). The interface is responsible for detecting the attempt to communicate to a Prolog computation the binding of a shared variable to a complex term during execution. However, such bindings can be passed at the beginning or at the end of execution, and the Prolog engine may make complex term bindings to shared variables at any time during execution of a \texttt{clp/1} (or \texttt{prolog/1} call). This semantic restriction is motivated by important implementation considerations. It is essential for solving a memory management problem with concurrent bindings in a canonical Prolog engine. It copes with the difficulty of where to put any complex terms which are transiently communicated...
from the CLP computation to the Prolog computation during the satisfaction of the concurrently executing Prolog goals. The place to build complex terms in a Prolog engine is on the top of the Prolog engine’s heap. However, complex term bindings to a shared variable by the CLP engine, which are transiently communicated from the CLP computation and stored at the top of the Prolog engine’s heap at the time of communication, would be vulnerable to being inappropriately popped after that time by local backtracking by sub-goals of the concurrently executing Prolog goals.

There is no other satisfactory place to put such bindings without instituting major changes to the design of the Prolog engine contrary to the minimal intervention coupling philosophy being espoused. So it is better to proscribe the transient making of complex term bindings to shared variables by the CLP computation, and to make the CLP-Prolog interface enforce that restriction. Bindings of simple terms to variables present no major difficulty, because typical Prolog implementations write them on top of the variable they are binding. Thus their bindings would be safe to such local backtracking, so long as these bindings are not trailed at the top of the trail stack at the time of communication. Instead a trail entry for each shared variable can be created at the time of initial execution of the \textit{clp/1} call, so that any bindings, which are communicated across, are undone should goal failure cause backtracking right back to the parent of the \textit{clp/1} goal. Despite the \textit{No Complex CLP Bindings} restriction it remains possible to define the lazy multiple solutions constructor \textit{subset/3}, which incrementally extends its solution list argument, using these CLP-Prolog interfaces as is shown in chapter 7. The solution depends on the tactic of using the code areas of the CLP and Prolog engines to pass solution list information both ways.

Two asynchronous interfaces allow a CLP computation to call Prolog as follows.

\begin{itemize}
  \item call to Prolog relation
    \begin{itemize}
      \item communication at boundaries
      \item executes concurrently with CLP resolution engine
      \item no suspension on invocation
    \end{itemize}
\end{itemize}
call to \textit{prolog}(Goals)

- communication transiently both ways (with a restriction)
- executes concurrently with CLP resolution engine
- input argument suspension if \textit{Goals} unbound

Goals in a CLP computation, which are not defined by a guarded clause relation and are not calls on a primitive, invoke the Prolog resolution engine. The values of the invocation are passed at the boundaries of invocation with full concurrency between the calling computation and the computation which is called. No suspension on invocation takes place. Calls in a CLP computation to \textit{prolog}/1 invoke the Prolog resolution engine asynchronously in a fully concurrent fashion. Input and output bindings are communicated transiently except that the \textit{No Complex Term CLP Bindings} restriction applies to the CLP computation executing concurrently with \textit{prolog}(Goals). The CLP computation is only allowed to bind shared variables in \textit{Goals} to simple terms when executing a \textit{prolog}/1 call. The interface is responsible for detecting the attempt to communicate to a Prolog computation shared variables bound to complex terms by the CLP computation. The call to \textit{prolog}/1 suspends on its argument, if it is unbound. \textit{prolog}/1 is rather similar to Clark and Gregory’s proposed interface \textit{prolog\_call}/1, although the implementation approach is quite different. Because \textit{prolog}/1 subsumes the functionality of \textit{set}/3 and \textit{subset}/3 as Clark and Gregory have shown, and as is shown by different means in chapter seven, multiple solutions interfaces like \textit{set}/3 and \textit{subset}/3 are not directly supported. Lower level mechanisms are required in the implementation of \textit{prolog}/1 and \textit{clp}/1.

### 3.6. Implementation Issues

There are various levels at which communication between the CLP and the Prolog engines can be achieved. High level interfaces will be described, because they demonstrate the functionality most readily. They prove that coupling is possible with low degrees of intervention into the mechanics of each engine. Invocation relationships will be further clarified by introducing two new primitives.

- \texttt{clp\_machine(Name, Goals, Result)}
- \texttt{prolog\_machine(Name, Goals)}

The first primitive is not resatisfiable and must only be invoked when \textit{Goals} is ground (apart from occurrences of \textit{Name}). Satisfying it, invokes the CLP engine, unifies \textit{Name} with a unique identifier of a file-like interface to the CLP engine, and starts executing \textit{Goals} in a separate meta-call environment concurrently
with any existing CLP engine goals. It associates Result internally with the CLP computation in such a way that the Prolog implementation unifies it with yes if and when the CLP computation ends successfully. `prolog_machine(Name, Goals)` is a CLP primitive which suspends until Goals is ground (apart from occurrences of Name). It then creates a new Prolog computation thread, unifies Name with a unique identifier of a file-like interface to it, and executes Goals in that thread. These primitives elicit the names of file-like interfaces which can be used to pass data between CLP and Prolog computations. These interfaces can be realised by pipes, sockets or shared memory buffers which are associated with these names. Buffered communication will be achieved between CLP and Prolog computations by reading and writing to these file-like objects in the same fashion as either would read or write to a file. A Prolog computation will be assumed to read input synchronously from a file (i.e. wait doing nothing until input arrives to satisfy a `read(Term, File)` primitive call). On the other hand CLP computations will be assumed to consume input asynchronously (i.e. suspend the primitive goal `read(Term, File)` and carry on executing other processes, or sleep if dead-locked, until input on File wakes up the suspended goal `read/2`).

In terms of Sun’s OS 4.0 version of Unix [Sun OS Manual 1988] multiple Prolog and CLP computations will interact as follows:

- CLP calls in Prolog invoke different CLP meta-calls in one CLP engine
- Prolog calls in different CLP meta-calls invoke different Unix processes
- Prolog calls in one CLP meta-call invoke different lightweight processes in a Unix process

CLP meta-calls mean Parlog-like stream controlled meta-calls [Gregory et al 1989] and not simple meta-calls like `call/1`. The CLP engine will be described in what follows as a single Unix process. However, it could also be realised as multiple tightly coupled Unix-like processes sharing memory. A Prolog computation will either be a single threaded ordinary Unix process with its own local memory, or be a lightweight Unix process within an ordinary Unix process which shares the memory of the Unix process with other lightweight processes (computation threads). Single threaded Unix processes executing Prolog will be standard WAM engines, capable of being forked into several lightweight processes which each maintain their own trail, stack and virtual heap, and use a common symbol table and code area. Separate threads synchronise their updates of the common memory like the code area and symbol table by using monitors on critical code. Other forms of concurrency control for integrity maintenance over the shared database are assumed to be achieved by explicit Prolog programming as explained in chapter six.
3.6.1. Prolog calling CLP Atomically

A simple synchronous invocation from a Prolog computation to a CLP computation by the goal \texttt{relation(Arg1, ..., Argn)} can be achieved by replacing it with the goal \texttt{guarded(relation(Arg1, ..., Argn))}, where \texttt{guarded/1} is defined by the following Prolog clause

\begin{verbatim}
guarded(Goal) :- clp_machine(Name, solve(Name), _),
write(Goal, Name),
read(Goal, Name).
\end{verbatim}

The primitive \texttt{clp_machine/3} has been defined above. \texttt{read(Term,File)} reads a \texttt{Term} from \texttt{File}. \texttt{write(Term,File)} writes a \texttt{Term} to \texttt{File}. \texttt{guarded/1} uses the following Parlog clauses:

\begin{verbatim}
mode solve(?).
solve(Name) :- read(Goals, Name),
call(Goals) | write(Goals, Name);
solve(Name) :- write('fail', Name).
\end{verbatim}

\texttt{read(Goals?, File?)} is a primitive which unifies \texttt{Goals} with the next term read from \texttt{File}. \texttt{call(Goals?)} is a primitive which suspends until its argument is bound before attempting to satisfy it. \texttt{write(Term?, File?)} is a primitive which suspends until its argument is bound before writing \texttt{Term} to \texttt{File}.

The Parlog goal \texttt{solve(Name)} with \texttt{Name} bound will be suspended on the two guard goals for the first clause for \texttt{solve/1} until satisfaction of the goal \texttt{guarded/1} succeeds in sending the term \texttt{relation(Arg1, ..., Argn)} to the CLP system. This will be received by the CLP engine and executed as a goal on the CLP engine by the primitive \texttt{call/1}. If \texttt{call(Goals)} fails, a term signaling failure is written back to the Prolog system, otherwise the satisfied goal is written back. The Prolog system is meanwhile waiting to read the output from the CLP system. When the Prolog system reads the term output as a result of the CLP execution, it attempts to unify the result with the original goal. The original Prolog goal succeeds or fails with that unification.
3.6.2. Prolog calling CLP Incrementally

The concurrent incrementally communicating interface \texttt{clp/1} is more complex to define. It needs a new Prolog primitive \texttt{async(Name,Goal)} for handling communication with the CLP computation. This primitive behaves like the \texttt{freeze(X,Goal)} primitive [Colmerauer et al 1983] except that instead of \texttt{Goal} being woken by the variable \texttt{X} being instantiated, \texttt{Goal} is woken by input being received by the Prolog computation from the file \texttt{Name}. Its other special characteristics are that each time it is woken up, its \texttt{Goal} is executed but it remains suspended, and is only removed if its point of creation is backtracked across. Furthermore delayed goal scheduling ensures that only one goal delayed by an \texttt{async/2} call may be woken and reduced at a time.

The primitive \texttt{async/2} can be efficiently implemented on WAM engines along the same lines as \texttt{freeze/2} [Carlsson 1987]. The behaviour of \texttt{clp/1} can now be supported at a high level by transforming rule bodies in which \texttt{clp/1} occurs. If a \texttt{clp/1} goal occurs in a rule body as follows

\begin{verbatim}
relation(X, Y, Z) :- try(Y, V), clp(test(X, V)), check(Z, V).
\end{verbatim}

then it can be replaced by the rule

\begin{verbatim}
relation(X, Y, Z) :- clp_call(test(X, V), Result), try(Y, V), check(Z, V), data(Result).
\end{verbatim}

\texttt{data/1} is a primitive which makes Prolog execution delay until its argument is bound. It ensures that the Prolog computation waits upon the CLP computation finishing. The relation \texttt{clp_call/2} is defined in Prolog as follows:

\begin{verbatim}
clp_call(Goals, Result) :- clp_machine(Name, invoke(Name), Result), 
 otherwise(Name, fail), 
 async(Name, talk(Goals, Name)), 
 write(Goals, Name).
\end{verbatim}

On executing \texttt{clp_call(Goals, Result)}, the shared variables in its arguments are trailed, and the goal is reduced to its body goals. The first body goal \texttt{clp_machine/3} creates an interface to the CLP engine, where the clauses which define the Parlog relation \texttt{invoke/1} are assumed to be defined in the CLP machine’s database already. The variable \texttt{Result} is also noted by the implementation for unifying with yes should the CLP computation succeed. An escape exception handler \texttt{otherwise/2} for the exception \texttt{Name} is then set to catch a failure in the execution of the \texttt{Goals} on the CLP engine, which will be handled by raising the exception \texttt{Name}. Escape exception handling is discussed in more detail in chapter six. It is instigated by
executing the special exception raising primitive `fail/1` and starts deep backtracking at one go across many
choice-points to the choice-point labelled by the argument of `fail/1` - in this case it will be the one set up by
executing `otherwise/2`. The alternative branch (i.e. the second argument of `otherwise/2`) is then executed.
The escape exception handler ensures that if the interface restrictions on not rescinding bindings to shared
variables are violated, an escape exception can be raised. The handler will catch the exception after undoing
as much of the concurrent computation, including any lingering choice points left by executing the
other goals in the body of a rule in which `clp_call/2` occurs, as has been done already. It allows this exceptional
circumstance to be gracefully transformed into simple failure of all the goals in the rule body. After
the escape exception handler is set up, a persisting delayed goal `talk/2` is created using the primitive `async/2`
explained above for handling communication with the CLP engine. The goal to be executed on the CLP
ingine is then sent to that engine by `write(Goals, Name)`, and Prolog execution continues with the Prolog
goals in the body of the Prolog rule in which `clp_call/2` occurs, which are to run concurrently with the call
to the CLP engine. In the `relation/3` example above, they are the goals `try/2` and `check/2`.

Each interface between CLP and Prolog computations maintains its own private library as to which computa-
tion is responsible for which bindings to terms. This library is used to enforce the *Irrevocable Communication*
restriction. Transient communication is handled by the persisting suspended goal `talk/2` defined as follows:

```prolog
talk(Name, Goals) :- prolog_bindings(Goals, Name),
read_goal(Goals, Name),
clp_bindings(Goals, Name),
write(Goals, Name), !.
talk(Name, _) :- fail(Name).
```

Two primitives `prolog_bindings/2` and `clp_bindings/2` record in their private library associated with the
interface `Name` any unrecorded bindings to variables in their first argument. Each library entry associates
the binding with the Prolog or the CLP side of the `Name` interface responsible for it. If `prolog_bindings/2`
notices that recorded Prolog bindings have been rescinded, it fails. Otherwise both predicates succeed. The
communication from the CLP engine is read and `Goals` is unified with it using a special primitive
`read_goal/2`. `read_goal(Goals, Name)` only succeeds if `Goals` can unify with the term read, and is able to
do so without binding variables in `Goals` to complex terms (*No Complex Term CLP Binding* restriction).
Furthermore if `read_goal/2` can succeed, it unifies its first argument with the term read, performing any
bindings it makes without trailing them on the trail stack by copying simple terms on top of variables. 

*Goals* is then written to the CLP computation again. Should any of this fail, an escape exception is raised by the second clause for *talk/2*. This results in the handler in *clp_call/2* catching it and then failing execution back to the original goal past any choice points left by the execution of the Prolog goals running concurrently with the CLP goal. In our case the original goal is the one which invoked the clause for *relation/3*, and its resatisfaction would result in an alternative satisfier being sought.

The CLP end of this interaction can be written in the contemporary variant of Parlog, where head argument matching strictly precedes guard satisfaction. The Parlog clauses for *invoke/1* assume the existence of a stream controlled meta-call

\[
\text{meta}(G?, R?, S^*)
\]

for invoking the CLP resolution engine concurrently with other invocations. This CLP primitive executes its *G* argument on the CLP engine for one quota of resources for every *quota* element on its *R* resources stream. It either reports using its *S* argument as a status stream each time it has used up a quota by *used_quota*, or reports that it has *succeeded, failed, woken* or *deadlocked*. In the event of having run out of resource quotas, *meta(G,R,S)* suspends on the variable tail of its resources stream *R*. In the event of a deadlock, *meta(G,R,S)* suspends on all top level variables in *G*. If any of these are instantiated, or if its meta-environment receives a *wake-up* inter-process signal from a Prolog computation to which it is interfaced, *meta/3* wakes up and adds a *woken* token to its status stream. It then waits for the next persisting delayed goal communication to complete on the associated Prolog computation, and adds a new *deadlocked* token to its *Status* stream, if the Prolog computation remains deadlocked.

```prolog
mode invoke(?).
invoke(Name) :-
  read(G, Name),
  meta(G, [quota|L], S),
  control(G, Name, L, S).
```
mode control(?, ?, ?, ?).
control(G, Name, L, [deadlocked|S]) :- write(G, Name),
                           read(G, Name) |
                           control(G, Name, L, S).
control(G, Name, L, [woken|S])  :- write(G, Name),
                               read(G, Name) |
                               control(G, Name, L, S).
control(G, Name, L, [succeeded|S]) :- write(G, Name).
control(G, Name, L, [used_quota|S]) :- write(G, Name),
                                 read(G, Name) |
                                 L = [quota|Ls],
                                 control(G, Name, Ls, S) ;
control(G, Name, L, [_|S])      :- write('fail', Name) |
                                 fail.
Transient communication with the CLP engine is achieved by writing and reading back the current values of goals and unifying the terms read with their own version of the goal. This is done each time the CLP computation deadlocks, or executes a quota of resources, or its control meta-call meta/3 is woken. The coupled system copes with mutual deadlocks, when a Prolog computation discovers it is deadlocked, by having the Prolog computation pass wake-up inter-process signals to processes with which it is interfaced. This continues on a periodic basis until the Prolog computation is itself woken up again. These wake-up signals are ignored by CLP computations unless they have deadlocked meta/3 meta-calls.

3.6.3. CLP calling Prolog Atomically

A simple call to a Prolog relation in the CLP program, using the CLP-Prolog scheme for combining CLP and Prolog execution, passes values at the beginning and end of its invocation. Thus the reduction of the goal is an atomic operation from the point of view of the CLP computation. If the CLP computation is executing Parlog and invokes the Prolog relation relation/n as a goal, then the invoked Prolog goal can be replaced by the Parlog goal

horn(relation(Arg1, ..., Argn)).

The relation horn/1 can be defined in Parlog as follows
mode horn(?).

horn(Goal) :- prolog_machine(Name, solve(Name)),
                      write(Goal, Name) | 
                      read(Goal, Name).

A Prolog computation thread is established, the Goal to be executed on it is written to the Prolog system, and the result is read back from the Prolog system and unified with Goal. The Prolog clause solve/1 can be defined as follows:

solve(Name) :- read(Goal, Name),
                 call(Goal),
                 write(Goal, Name).

solve(Name) :- write('fail', Name).

The goal is read, executed, and either written back if successful, or a reserved atom which cannot unify with the goal is written back instead.

3.6.4. CLP calling Prolog Incrementally

The incrementally communicating interface prolog/1 is rather more complex to realise. In order to support transient communication the Prolog engine must send and receive at appropriate junctures the current bindings of the common goal. In order to achieve this a stream controlled primitive

wam_machine(Nameˆ, Call?, Resources?, Statusˆ)

is introduced for creating and scheduling execution of a Prolog engine. This CLP primitive suspends until Call is ground apart from occurrences of the variable Name and until Resources is bound. It then creates a new Prolog computation thread, unifies Name with the unique identifier of a file-like interface to it, and executes the goal Call in that computation thread for one quota of reduction steps for every quota element on its Resources stream. It either reports using its Status argument as a stream each time it has used up a quota by used_quota, or reports that it has succeeded, deadlocked, or failed. In the event of a deadlock, which is not cleared merely by the execution of goals delayed by async/2, the wam_machine/4 goal suspends pending further activity in the Prolog engine bar persisting delayed goal execution. In the event of having run out of resources, it suspends on the tail of its resources stream R. Further resources can be given to the Prolog engine by appending a new quota element to the end of its Resources stream, allowing the
Prolog computation to continue. Another primitive required is

\[
\text{write\_and\_wait(Term?, Name?, Term1?)}
\]

It executes atomically by writing `Term` to the file `Name` and suspending on all the variables in `Term1`. It succeeds if and when a variable in `Term1` is bound. These primitives can be used to define `prolog/1` in the variant of Parlog, where head argument matching strictly precedes guard satisfaction, as follows:

```prolog
mode prolog(?).
prolog(Goals) :- wam_machine(Name, resolve(Name), [quota|Cs], S),
               write(Goals, Name),
               manage(Goals, Name, Cs, S).

mode manage(?, ?, ˆ, ?).
manage(Goals, Name, Cs, [used_quota|Ss]) :- write(Goals, Name),
                                        read(Goals, Name) |
                                        Cs = [quota|Cs1],
                                        manage(Goals, Name, Cs1, Ss).
manage(Goals, Name, Cs, [succeeded|Ss]) :- read(Goals, Name).
manage(Goals, Name, Cs, [deadlocked|Ss]) :- handle_deadlock(Goals, Name, Ss) |
                                       manage(Goals, Name, Cs, Ss) ;
manage(Goals, Name, Cs, [_|Ss]) :- write('fail', Name).

handle_deadlock(Goals, Name, Ss) :- bound(Ss) |
                                  true;
handle_deadlock(Goals, Name, Ss) :- write_and_wait(Goals, Name, (Goals, Ss)),
                                 read(Goals, Name) |
                                 handle_deadlock(Goals, Name, Ss).
```

`prolog/1` creates a new Prolog computation thread using `wam_machine/4`. It writes `Goals` to it, and then lets the concurrent goals for the relations `wam_machine/4` and `manage/4` continue the transient communication. Whenever the computation uses a quota, deadlocks, wakes, or succeeds, the CLP computation receives notification of this on the status stream of the `wam_machine/4` goal. The CLP computation then sends to the Prolog computation its current version of the Prolog goal, and reads the CLP computation’s current version of the goal. `handle_deadlock/3` is used to handle deadlocks on the Prolog computation. It uses the time of call primitive `bound/1` to find the unbound end of the stream, and waits on receiving a message on the `Status` stream or failing that on further instantiations of `Goals`, passing `Goals` back and forth with each instantiation until the deadlock is cleared.
The Prolog code for this interface is as follows:

```prolog
resolve(Name) :- read(Goals, Name),
               async(Name, talk(Goals, Name)),
               call(Goals),
               write(Goals, Name).
resolve(Name) :- write('fail', Name),
```

`resolve/1` reads in `Goals`, sets up a communication link with the CLP computation using `talk/2` in the fashion explained before, tries to satisfy `Goals`, and writes the final version of the goal `Goals` back to the CLP computation.

### 3.7. Conclusion

A deterministic coupling of Prolog and CLP resolution engines, enables the use of trial bindings in a don’t know non-deterministic search to be safely isolated and hidden within a Prolog sub-computation. This allows CLP resolution to sustain a system of concurrent communicating tasks without risk of uncommitted bindings rolling back the history of the system interaction in an unwanted fashion. Four interfaces between coupled CLP and Prolog computations have been defined, and a simple non-shared memory implementation scheme on single and multiple processor machines has been proposed. A concise CLP-Prolog meta-interpreter for all four interfaces is given in chapter seven. It helps prove the simplicity and power of the interfaces. Coupled resolution enables multiple mutually invoking CLP and Prolog computations to be made concurrently. The effect is to achieve a coupled logic programming system which can manage systems programming tasks on the CLP engine, and deductive querying on the Prolog engine. It can support multiple concurrent activities in each area at the same time. Concurrency is partly realised through the Unix operating system among the CLP engine and various tightly or loosely coupled Prolog processes. It is also partly realised through the concurrency handling mechanisms of the CLP engine.
Chapter 4

4. A Lingua Franca of CLP Languages

4.1. Introduction

Several concurrent logic programming languages have been proposed. They all employ committed choice resolution and support the process interpretation of systems programming to some degree. This chapter considers their merits in the search for an expressive CLP language which can be readily implemented to execute on the CLP engine part of the coupled resolution engine system. Two of the most important non-flat CLP languages, GHC and Parlog, balance the requirements of clean semantics and good control rather differently, and their respective merits are compared and contrasted. Since concurrent logic programming would benefit from both, a lingua franca of these languages is characterised. A method for translating GHC and Parlog to and from it is given which preserves the arities and execution conditions of each clause. It enables an implementation of the lingua franca to support both languages transparently. The lingua franca also has merits as a CLP language suitable for programming in its own right.

4.2. GHC and Parlog

Among well-known CLP languages like Concurrent Prolog [Shapiro 1983, CP [Saraswat 1987a] and P-Prolog [Yang & Aiso 1986], two of the most prominent are GHC [Ueda 1986a] and Parlog [Gregory 1987]. These languages are or can be expressed in the same general syntax as Prolog [Clocksin & Mellish 1981]. See Appendix A for a full BNF description. A program in GHC and Parlog is a set of relations \( <R_1, ..., R_n> \). Each relation \( R_i \) is composed of guarded Horn clauses with the same predicate name and arity. Each clause has the form

\[
H \leftarrow G_1 \text{ AND } ... \text{ AND } G_m \mid B_1 \text{ AND } ... \text{ AND } B_n
\]

where \( H, G_1, ..., G_m, B_1, ..., B_n \) are atomic formulae. The clause head \( H \) gives the clause’s relation name and arity. The \( G_j \)s are the guard goals and the \( B_j \)s are the body goals. The commitment operator | separates the guard goals from the body goals. AND is a meta-symbol signifying a conjunction operator. The declarative reading of this clause is
H is true if \( G_1 \) and ... and \( G_m \) and \( B_1 \) and ... and \( B_n \) are true

In GHC AND is a place filler for the parallel conjunction operator ",". In Parlog it is a place filler for either the parallel conjunction operator "," or the sequential conjunction operator "&". One or more guarded Horn clauses form a relation

\[ C_1 \text{ OR ... OR } C_n. \]

where each \( C_i \) is a guarded Horn clause, OR is a meta-symbol acting as a place filler for a clause search operator, and the symbol "," terminates the relation. In GHC OR is only a place filler for the parallel search operator ",". In Parlog OR is a place filler for either the sequential clause search operator ";" or the parallel clause search operator ",". These operators are of equal precedence and are right associative. Parlog clauses for the same relation \( R \) of arity \( n \) can be preceded by a single mode declaration

\[
\text{mode } R(M_1, ..., M_n).
\]

for that relation where each \( M_i \) is either the input argument symbol ? or the output argument symbol ^. All the head arguments of Parlog clauses for the same relation are classified as input or output by this declaration. GHC and unmoded Parlog clause head arguments are always classified as input arguments.

It is possible to redescribe the operational semantics of GHC in such a way that its correspondences with Parlog come out. From this point of view, to execute a GHC and Parlog program is to refute a conjunction of goals by input resolution as follows:

\[ L_1 \text{ AND ... AND } L_n \text{ where } n \geq 1 \]

Ignoring for the moment Parlog’s sequential search and conjunction operators, each goal is solved in parallel by being matched on all input arguments of clause heads for that relation. This means determining in parallel whether the relevant arguments of the goal can be unified with the input head arguments of the clause without instantiating or sharing the variables of the goal. Whichever clauses are able to meet the head argument matching requirements, have the input head arguments of a fresh copy of their clause unified with corresponding arguments of the goal. Other clauses which might satisfy the head argument matching requirements, if the goal was instantiated further, are suspended upon relevant goal variables. They have their input head arguments unified with corresponding goal arguments, if and when the variables upon which they are suspended are instantiated. In parallel with head argument matching, guard goals for each clause are solved in parallel. If one or more clause’s head argument matching requirements and guard goals
for that relation are satisfied, one of these is selected non-deterministically. Parallel clause searches are terminated and the goal is reduced to the goals in the body of that clause. If a guard goal or head match fails for each clause of the goal’s relation, the goal fails†. If this failed goal is part of a resolvent obtained from the original goals then the whole computation fails.

Parlog execution is subject to the following special conditions:

- output unification on head arguments
- sequential clause search
- sequential literal conjunction

Upon reduction, the output arguments of Parlog clause heads, as specified in the mode declaration, are unified with their corresponding goal arguments. Where Parlog clauses have sequential search restrictions on clauses, then a goal only initiates guard goal satisfaction, head argument matching and then argument unification with all clauses up to the next unencountered sequential search clause terminator. Only where a guard goal or a head match fails in each such clause, are the heads and guards of clauses up to the next unencountered clause search operator evaluated. If Parlog literals are conjoined by the sequential conjunction operator "&", then only when the goal or conjunction of goals on the left hand side of the operator has been satisfied, is satisfaction attempted on the right hand side goal or goal conjunction.

GHC execution is subject to the following special conditions:

- synchronisation rule
- sequencing rule

The synchronisation rule states that the attempt to bind a calling argument variable in the passive part (head and guard) prior to commitment should cause the unification to suspend. It guarantees safety for guard evaluation, namely that a guard whose clause does not figure in the reduction of a goal will not instantiate the goal. The sequencing rule states that the body of a clause may be executed before commitment, so long as the attempt to bind a guard, head or calling argument variable prior to commitment causes the unification to suspend. It can be trivially satisfied by delaying execution of body goals until after commitment.

† GHC is actually defined without assuming finite failure, but effective implementation make assuming finite failure a practical necessity. See chapter 4 especially section 4.10 [Ueda 1986a] and [Saraswat 1987a] p.352-353.
A distinctive feature of GHC and Parlog is that parallel guard evaluations are performed in a single binding environment. Some other committed choice languages like Concurrent Prolog [Shapiro 1983] and CP(↓,↑) [Saraswat 1987a] require that their guards be evaluated in separate environments. However, the extra complexity involved in supporting multiple guard environments for their execution makes them less than promising prospects for implementation. Flat variants of committed choice languages like Flat Concurrent Prolog [Shapiro 1986], FGHC [Sato et al 1987] and Flat Parlog [Foster & Taylor 1987] evade operational complexity in supporting guard evaluation by only allowing primitives to appear in clause guards. However, the approach adopted here is to stick with the rather more expressive full versions of committed choice languages. What is argued here, applies mutatis mutandis to those fragments of the full language which can be translated down to the flat subsets [Levy & Shapiro 1987].

As GHC and Parlog have rather similar execution models but complementary virtues, it would seem desirable to find some common denominator of both languages, which is on the one hand sufficiently powerful to be worth programming in its own right, and yet is on the other hand sufficiently close to both languages to be inter-translatable with them. Such a language could be used to support both the programming styles of GHC and Parlog, as well as supporting its own common denominator style of programming. If this lingua franca was simpler to implement than either GHC and Parlog, then supporting it directly could harvest the virtues of both GHC and Parlog as well as reaping the benefits of a more streamlined implementation. The purpose of this chapter is to present such a lingua franca.

A preliminary summary of its details is given to clarify future discussion. Appendix A gives a common BNF description of the syntax of the lingua franca, GHC and Parlog. The lingua franca should be thought of as a common variation of both GHC and Parlog with very similar syntax and closely related semantics rather than as a new language. Like both GHC and Parlog a lingua franca program is a set of relations <R₁, ...,Rₙ>. Each relation Rᵢ is composed of guarded Horn clauses

\[ H : G₁ \text{ AND } ... \text{ AND } Gₘ | B₁ \text{ AND } ... \text{ AND } Bₙ \]

where AND signifies a parallel conjunction operator ",". Like GHC the lingua franca has no sequential conjunction operator. These clauses Cᵢ form relations

\[ C₁ \text{ OR } ... \text{ OR } Cₙ. \]
In a relation each $C_i$ is a guarded Horn clause, and the symbol "." terminates the relation. Like Parlog OR is a meta-symbol acting as a place filler for either a sequential clause search operator ";" or a parallel search operator ".". Lingua franca relations are modeless in the fashion of GHC with each clause head argument occurring in an input matching role. Lingua franca goals are executed by and-parallel input resolution in the common fashion described earlier for GHC and Parlog except that sequential clause search operators constrain or-parallel execution in the Parlog style. The point of developing a lingua franca of GHC and Parlog is established by considering the factors responsible for the differences between Parlog and GHC.

4.3. The Control Emphasis

Parlog thematises the control of logic programs and is full of constructs to allow the programmer to synchronise and sequence execution of parts of a logic program. Parlog supports both sequential and parallel operators for clause search and conjunction. Thus the following clauses for `process/1` for sequential processing of a list of query/response tuples

```prolog
mode process(?).
process([]).
process([message(Q, R)|B]) :- valid_query(Q) | execute(Q, R) & process(B) ;
process([_|B]) :- true | process(B).
```

recursively take a list of message elements, and test whether the query part $Q$ is a valid goal. If it is, it is executed yielding a response $R$. Otherwise the message element is ignored. The sequential conjunction operator in the second clause body ensures that each element is executed before processing continues on the next element. The second clause’s sequential search operator ensures that only failure of the `valid_query/1` guard test allows the third clause to be tried. Both sequential operators enhance the control features of Parlog. They can be used to sequence calls to primitives for input and output, to help control the extent to which computation is demand driven, and to control the granularity of parallelism.

Parlog’s and-sequential and or-sequential operators have an operational meaning, but their sequentiality does not make any contribution to the declarative meaning of Parlog’s clauses and queries. The presence of these sequential operators affect the termination properties of Parlog programs. Swapping sequential conjunction operators for parallel conjunction operators can cause the deadlock of Parlog goals which used to succeed. Thus the goal `valid(A), A = truth` succeeds over the clause
mode valid(?).
valid(truth).

whereas the sequential variant valid(A) & A = truth suspends for ever. Although the operator swap makes
no change to the declarative meaning of the query, it changes the operational meaning of the query so that
the emulator is no longer able to establish the query’s deducibility. Thus the sequential operator & intro-
duces a way in which Parlog’s operational semantics can fail to complement its declarative semantics.

The lack of emphasis on clean semantic properties for Parlog and the objective of not making Parlog diffi-
cult to implement resulted in Parlog being defined so as to make it possible for certain kinds of Parlog pro-
grams, those with unsafe guards, to be invalidly executed. Unsafe guards can sabotage the successful
reduction of a goal by relevant clauses, by enabling irrelevant guard evaluations, whose clause does not fig-
ure in the reduction of a goal, to bind calling arguments. For example let some clauses for a relation
choice/2 be defined as follows:

mode choice(? , ?).
choice(A, life) :- way(A) | true.
choice(B, faith) :- true | B = true.

mode way( ).
way(C) :- true | C = godless.

Let the call choice(X, Y), Y = faith be made. It is now possible for the following things to happen

- X in the first top level goal is shared with the first argument A of the first clause for choice/2
- X in the first clause’s guard is bound to godless by satisfaction of way(X)
- Y in the second top level goal is bound to faith
- the first top level goal’s head match on the first clause fails

all before the first top level goal’s match on the second clause head succeeds and instantiates B to godless.

If all this happens, then the top levels goals are reduced to the body of the second clause for choice/2 - god-
less = true - which fails. Thus the examination of a subsequently rejected clause has bound X to godless,
which has interfered with the top level goals being able to succeed as they should.

In order to try to rectify this kind of difficulty, a compile time safety check has been proposed to analyse
whether Parlog guards are unsafe. Unfortunately analysing the safety of a Parlog program at compile time
is an undecidable matter in general [Gregory 1987] p.121-132. Thus there is no algorithm for selecting out all and only those Parlog clauses in Parlog programs whose guards are unsafe. This semantic anomalousness is intrinsic to the language itself.

The control emphasis of Parlog also allows Parlog to use primitives like `var/1` which have time of call semantics [Gregory 1987] p.82. Thus the conjunction of goals called with A unbound

\[
\text{var}(A), \ A = \text{true}
\]

succeeds or fails depending upon whether the first primitive is executed before the second. Like unsafe guards, this time of call behaviour makes it difficult to prove correctness properties of programs, because whether a goal succeeds or fails depends not upon the meaning of the goal but upon non-deterministic properties of the implementation. Time of call properties are not confined to odd primitives like `var/1`. They apply even to key Parlog primitives like the two argument meta-call `call/2`. This primitive, which always succeeds, initiates the attempt to satisfy its first argument and unifies its second argument with a constant expressing the result. However, there are examples of its use [Ueda 1986a] 4.7.2.

\[
\text{call}(X = 1, _), \ X = 2
\]

exhibiting time of call semantics. The time of execution of each conjunct in this example determines whether the conjunction succeeds or fails.

Parlog is also furnished with side effect causing primitives for input and output in the manner of Prolog [Gregory 1987] p.147. A semantic difficulty here is that with the possibility of such primitives being scattered across multiple clauses the ordering of such side effects becomes a non-local property of the execution of such a set of clauses. This ordering property is not required to be reflected in the declarative semantics of this set of clauses. Thus Parlog’s semantic drawbacks are that

- its sequential operators create execution order constraints not reflected in Parlog clauses’ declarative semantics
- unsafe guards cannot be precisely identified to prevent them from causing invalid execution
- primitives with time of call semantics make computation outcomes non-deterministic
- input and output primitives have side effects whose orderings are not embodied in their clauses’ declarative readings
These prices are paid in addition to the basic semantic price paid by all CLP languages in precluding complete searches through embracing the don’t care non-determinism of the committed choice mechanism. However, these extra semantic deficiencies have been bought at the significant price of enhancing control in Parlog and it is these features which make it so apt for systems programming [Foster 1988].

4.4. The Semantic Emphasis

GHC approaches CLP language design from the opposite point of view to Parlog. Instead of thematising control issues, it thematises semantic issues in CLP language programs. Sequential conjunction and search operators are excluded from the language to remove their execution order constraints from marring the declarative reading of guarded clauses and the termination properties of programs. The rule of synchronisation is introduced to ensure that guards are guaranteed to be safe. Primitives like var/1, ==/2 and meta-calls are excluded to eschew time of call effects. Input and output primitives are made more declarative by outlawing their isolated occurrence and by drawing together their action instead as list elements in calls to a single meta-primitive instream/1.

    instream( [ write('What? '), read(Response) | Tail ] )

Goals on this stream are executed in sequence as they become sufficiently instantiated. Thus satisfying this primitive instantiates its argument to an ordered list of input and output goals. Lastly by localising the generation of all output bindings to the unification primitive =/2, GHC is able to conform to a powerful unification decomposition principle called anti-substitutability [Ueda 1986a] 4.7.2. The effect of all these measures is to produce a simple and powerful language which is open to the application of a variety of transformation techniques [Ueda 1986b, Fujita, 1988], and is amenable to formal analysis [Ueda 1985, Saraswat 1987a, Kanamori & Maeji 1988].

However, this emphasis of GHC on the clarity and simplicity of its semantics weakens its ability to be used as a general purpose programming tool. By excluding primitives like var/1, GHC is unable to test directly whether a variable is instantiated at the time of call or not, even though it can synchronise execution of goals on that conditionality. This precludes GHC from defining unification within itself, and also precludes it from being able to define a wide range of related unification functions within itself. The ability to program variations on unification is crucial to a logic programming language’s ability to program meta-interpreters of languages and language flavours different from itself. Experience with Prolog has demonstrated
that Prolog’s ability to support a wide variety of meta-interpreters of related languages on top of itself is a major part of the reason for its success [Sterling & Beer 1989]. By losing access to the meta-programming of unification, GHC has hamstrung its capacity to mirror Prolog’s wide meta-programming capability. Giving GHC a primitive like \texttt{var/1} would not be enough. GHC would also need some means for ensuring atomicity for unification related operations defined using \texttt{var/1}. One possibility might be to make \texttt{var/1} a micro-instruction of a substrate language for defining GHC primitives. \texttt{var/1} would remain hidden, but its functionality could be exploited without violating GHC’s semantic requirements.

Systems programming in GHC-like languages requires the ability to handle failures, exceptions, and runtime errors of a sub-computation in a modular fashion which localises their effects. It also requires the ability to control and interact with them as tasks - to suspend, resume or abort them. Flavoured meta-interpreters or meta-interpreters with extra control features are needed for this kind of job. They can either be explicitly programmed or implemented by means of control meta-calls [Foster 1987]. Real operating systems also need dynamic and programmable control over scheduling and resource allocation to sub-computations which only a sophisticated form of control meta-call can achieve [Foster 1988]. However, as GHC refuses to support Parlog style meta-calls and refuses to support primitives needed for programming unification functions, GHC is limited in its ability to support flavoured meta-interpreters. The result is an impoverished systems programming capability†.

GHC’s eschewal of sequential search operators makes programming in GHC more difficult. For example, lack of a sequential search operator makes it more difficult to control the grain of or-parallelism by making it harder to control the order of clause examination. Furthermore where a clause choice is made between a user-defined condition \texttt{test/1} holding and its not holding,

\begin{verbatim}
goal(In, Out) :- test(In) | process(In, Out) ;
goal(In, Out) :- true   | transform(In, Out).
\end{verbatim}

it is very convenient to express this directly by sequential search rather than by using negation as failure \texttt{not/1} [Gregory 1987] p.89-92.

† The design of KL1-B, the basic machine language for ICOT’s parallel inference machine [Sato et al 1987], has already been influenced by these considerations. KL1-B [Kimura & Chikayama 1987] is based upon Flat GHC but has been extended by pragmas and a meta-call. The price is the loss of GHC’s nice semantic properties.
goal(In, Out) :- test(In) | process(In, Out).
goal(In, Out) :- not(test(In)) | transform(In, Out).

which in the event of the condition not holding might result in the condition being evaluated twice. It is noteworthy that all other well-known committed choice languages like FCP, Parlog, CP and P-Prolog support sequential search in some form. Because GHC lacks a sequential conjunction operator, GHC programs require much more transformation to be able to exploit the short circuit technique for testing when a set of goals has all been satisfied than Parlog programs. Using this technique a set of goals

\[
\text{alpha}(A, D), \text{beta}(A, B), \text{gamma}(C, D), \text{delta}(D, B).
\]

is transformed to give each goal two extra arguments in a chain.

\[
\text{alpha}(A, D, X1, X2), \text{beta}(A, B, X2, X3), \text{gamma}(C, D, X3, X4), \text{delta}(D, B, X4, X5).
\]

As each goal terminates, the last thing it does is to unify its last two arguments. The test for joint termination is done by binding one end \(X1\) to some token like \texttt{done} and waiting until the other end of the chain \(X5\) gets bound to \texttt{done}. Because all GHC conjunctions are executed in and-parallel, GHC programs cannot use Parlog’s simple expedient of sequencing the execution of the goal with the unification of the local chain ends.

\[
\text{mode alpha}(?, ?, ?, ?).
\]

\[
\text{alpha}(A, D, X1, X2) :- \text{alpha}(A, D) \& X1 = X2.
\]

Neither can GHC generally use its sequencing rule

\[
\text{alpha}(A, D, X1, X2) :- \text{alpha}(A, D) \mid X1 = X2.
\]

to achieve the same effect, because such sequencing is always subject to the requirement for preserving guard safety. If the second argument of \textit{alpha}/2 accepted input but tried to instantiate it further then the goal \textit{alpha}(A, D) would suspend on that attempted binding. To use the short circuit technique full generally GHC programs must be extended to test for the termination of each primitive call liable to bind variables in the relevant program! For each unification \(A = B\) in the body of the GHC clauses defining \textit{alpha}/2 this means replacing it with the goal \textit{unify}(A, B, X1, X2) defined as follows [Shapiro 1989] p.462-463.
The local part of the short circuit is only closed when it has been detected that the unification has been performed. Plainly the overheads of such a transformation mark a distinct gulf of expressiveness between GHC and Parlog. See [Shapiro 1989] p.448ff, 462-463 for more discussion of the short circuit technique.

This chapter shows that there is no need to make a choice between either having a CLP language with clean semantics like GHC or having one with a good systems programming capability like Parlog. It is possible to benefit from the respective virtues of each language on a single implementation by rewriting each of them in a lingua franca and emulating this language directly.

### 4.5. The Common Linguistic Denominator

While GHC and Parlog share the common characteristic of being committed choice languages which evaluate primitive and user defined guards for rival clauses in a single environment, they each have individual features specific to their own language. Plainly a committed choice language variant of GHC and Parlog which aspires to be a lingua franca to them must abstract from these differences and yet be such that both GHC and Parlog can be translated to this language and back again.

Besides special restrictions and requirements for primitives which will be discussed later, GHC has five main features which distinguish it from Parlog.

- no mode declarations
- purely parallel search operators
- purely parallel conjunction operators
- clause bodies are executable before commitment so long as trying to instantiate a body variable shared with the head or guard causes the unification to suspend until commitment
- an attempt by a guard literal to write upon a calling argument variable before commitment causes the unification to suspend

### 4.5.1. Mode Declarations

The absence of mode declarations makes GHC simpler than Parlog. GHC input matches on each head argument and explicitly represents output bindings with unifications in the bodies of clauses. Parlog can be
easily transformed to obey the same rule. Parlog head arguments which are declared as output arguments can be removed and replaced by a fresh variable and an extra unification between the fresh variable and the original head argument can be added to the body of the clause. This transformation is part of the translation of Parlog to Kernel Parlog [Gregory 1987] p.64-69. After this change all Parlog head arguments can be treated as input arguments and mode declarations can be dispensed with. This transformation possibility reveals a common feature between Parlog and GHC which can be copied by the lingua franca. In this way the lingua franca need never to have to support mode declarations at all.

4.5.2. Passive Part Concurrency

Both GHC and Parlog allow head argument matching to proceed in parallel with guard evaluation. The lingua franca would be made rather simpler, if it departed from this precept and required that head argument matching was always completed before guard evaluation began. This would make the lingua franca simpler to implement. It would eliminate having to cope with variables shared between head arguments and guard literals, which are accessed during the execution of a guard literal before the relevant binding for the variable can be supplied from the goal by a head argument match.

Where clause guards are empty (signified by true), then no transformation is needed in translating GHC or Parlog to the lingua franca to compensate for the lingua franca’s sequencing of head matching and guard evaluation. Furthermore in clauses where no guard goal can fail before the head argument match succeeds, no transformation is necessary either. However, in other cases the head argument match has to be performed in parallel with guard evaluation. Otherwise it would be possible for suspension in the head argument match to delay indefinitely discovery that execution of the guard goals will fail. Thus where the guard is non-empty and where it cannot be determined that the guard will not fail before the head argument match succeeds, the head argument match can be decomposed into an extra one way unification in the fashion for translating Parlog to Kernel Parlog [Gregory 1987] p.65-66. A variant of Gregory’s method uses a one-way unification primitive \( <=/2 \) which attempts to unify its arguments and suspends on relevant right hand argument variables if unification would instantiate one or share two of them. All head argument matches are compacted into one primitive call by forming an aggregate term out of each argument needing matching and performing the match against a similar aggregate of their replacements using a single primitive call. Thus the guarded clause
compare([A|B], [C|D]) :- match(A, C) | compare(B, D).

can be translated to

compare(E, F) :- ([A|B], [C|D]) <= (E, F), match(A, C) | compare(B, D).

Execution would not need to spawn separate processes to perform the head match on each argument as with Gregory’s method. However, <=/2 would need to incorporate subtle functionality for ensuring that calling argument variables were not shared. On multi-processor implementations contention for exclusive write access to variables might be exacerbated by tying together so many variables into one process.

### 4.5.3. And Sequential Operators

Unlike GHC, Parlog supports and-sequential operators. However, the synchronised satisfaction of Parlog goals can be achieved indirectly. Gregory eliminates sequentially conjoined literals from Parlog clauses using meta-calls and synchronisation flags linking the meta-calls [Gregory 1987] p. 98-99, 141-142. The absence of and-sequential operators in GHC and the possibility of eliminating sequential conjunction operators in Parlog in such a fashion establishes another common feature between both which can be copied by the lingua franca. It need only support and-parallel conjunctions. And-sequential operators can be eliminated by transforming them away from the Parlog source.

Rather than using Gregory’s method which employs Parlog’s three argument control meta-call, a similar but different three argument meta-call `wait/3` is introduced instead. It makes translation back easier to perform, because each occurrence of the new meta-call is known only to arise as a result of eliminating a sequential conjunction. It also requires no auxiliary relation to be defined, which creates a simpler translation product. The basic idea of translating all sequential conjunctions into parallel conjunctions, and controlling their execution order with synchronisation flags remains the same. Thus the mixed conjunction

\[
\text{head}(A, B, C) :- \text{true} \mid (\text{first}(A), \text{second}(B)) \& \text{third}(C).
\]

can be translated as follows:

\[
\text{head}(A, B, C) :- \text{true} \mid \text{wait(first}(A), [], D), \text{wait(second}(B), [], E), \text{wait(third}(C), [D, E], _).
\]

The new meta-call primitive `wait/3` suspends when executed until its second argument is ground. When ground, it initiates execution of the goal in its first argument. It fails, if this goal fails. If it succeeds, it
signals the result in its third argument by binding it to a constant.

The general form of this method for translating away sequential conjunctions applies to any Prolog syntax structure not containing the reserved predicate `wait/3`. Given the following query

```
|  ?- p(Goals, New_goals, [], _, parallel).
```

over the following pure Prolog program, where `Goals` is instantiated to a syntactically valid conjunction of literals, then `New_goals` will be unified with a purely parallel conjunction.

```
p((A, B), C1, Priors, [], parallel) :- p(A, A1, Priors, _, parallel),
p(B, B1, Priors, _, parallel),
join(A1, B1, C1).
p((A, B), C1, Priors, Results, sequential) :- p(A, A1, Priors, Results1, sequential),
p(B, B1, Priors, Results2, sequential),
append(Results1, Results2, Results),
join(A1, B1, C1).
p((A & B), C1, Priors, [], parallel) :- p(A, A1, Priors, Results1, sequential),
append(Results1, Priors, Priors1),
p(B, B1, Priors1, _, sequential),
join(A1, B1, C1).
p((A & B), C1, Priors, Results, sequential) :- p(A, A1, Priors, Results1, sequential),
append(Results1, Priors, Priors1),
p(B, B1, Priors1, Results2, sequential),
append(Priors1, Results2, Results),
join(A1, B1, C1).
p(A, wait(A, [], R), [], [R], sequential) :- literal(A).
p(A, wait(A, Priors, R), Priors, [R], _) :- literal(A),
Priors \= [].
join((A,B),C,(A,B,C)).
join(A,B,(A,B)) :- literal(A).
literal(A) :- A \= (_&_), A \= (_ & _).
```

The Prolog program more succinctly encapsulates the transformation algorithm than words.

### 4.5.4. Or-Sequential Search

GHC only supports or-parallel clause search, although the option of allowing sequential search by means of an `otherwise` predicate was considered in GHC’s design at one stage [Ueda 1986a] 3.4.7. On the other hand
Parlog allows clauses to be searched in parallel or in sequence. Gregory has shown that it is possible to eliminate or-sequential clause search using the three argument Parlog meta-call [Gregory 1987] p.140-141. However, using Gregory’s method would mean abandoning the principle of representing each Parlog or GHC clause with a lingua franca clause translation of the same relation name and arity. Fortunately, it is unnecessary, because it is fairly easily to support sequential search directly in CLP implementations as is shown in [Gregory et al 1989, Crammond 1988], and the implementation described in chapter five.

4.5.5. Rule of Sequencing

The option on executing the bodies of GHC clauses before commitment subject to GHC’s rule of sequencing [Ueda 1986a] may seem to be appropriate for highly parallel data-flow architectures, but is likely to result in a lot of redundant computation on less parallel architectures at the expense of what could be more profitably executed. Furthermore, by allowing such premature computation almost no GHC program can be guaranteed to terminate without importing extra fairness assumptions [Saraswat 1987a] p.353-354. To avoid these semantic difficulties and in keeping with extant implementations of GHC [Levy 1986a, Ichiyoshi et al 1987], no special suspension mechanism is proposed for supporting GHC’s rule of sequencing by the lingua franca. An approach to translating GHC clauses to the lingua franca will be adopted which never allows the evaluation of the body of a translated GHC clause to be concurrent with the evaluation of that clause’s guard. The same rule obeyed by Parlog of strict sequencing of evaluation will be obeyed by the lingua franca and only when the clause has committed will evaluation of the body commence. In this way the rule of sequencing can be observed by lingua franca translations of GHC clauses without special suspension functionality being provided.

4.5.6. Rule of Synchronisation

GHC has a rule of synchronisation which preserves guard safety by requiring that if a body unification would bind or share calling argument variables in an ancestor guard, that unification should suspend. Thus if the call ask(A) is made over the clauses

\[
\text{ask}(A) :- \text{tell}(A) \mid \text{true}.
\]
\[
\text{tell}(A) :- \text{true} \mid A = \text{yes}.
\]
then it should suspend, because the body unification $A = \text{yes}$ has to suspend in order not to instantiate a calling argument variable in an ancestor guard.

Two of the main approaches canvassed for implementing the guard suspension test are:

- Ueda’s Pointer Colouring Scheme [Ueda 1986a, Levy 1986a]
- Miyazaki’s Guard Numbering Scheme [Kishi et al 1985]

Both schemes concentrate upon suspending unifications in the bodies of GHC clauses being used to evaluate guard literals in the event of the unification attempting to write upon a calling argument variable. Either scheme threatens to impose a suspension test overhead upon every explicit unification. However, most GHC clauses either only match input against head arguments or have only system primitives with input arguments for guards. Only in evaluating the remaining cases is it necessary to test for a user defined guard literal or system primitive with output arguments writing upon a calling argument variable. A more demand driven scheme would separate out the suspension producing conditions from GHC clauses, in order to localise the requirement to suspend to evaluation of an extra system literal in the responsible guard. This would impose no general overhead upon unification in the body. By allowing unsafe guard literals to be evaluated without suspension constraints on their output bindings, evaluation is made more eager and computation space is more effectively recycled. Computation space used in evaluating the guard literal is claimed and released as soon as the guard literal can be satisfied or fails. It is not claimed and then frozen unused pending the further communication of input values to remove from suspension a suspended unification. For deep guard evaluations this frozen space may include space claimed to store ancestor processes suspended upon the unification process as well as that claimed to store the unification process.

Levy and Shapiro [Levy & Shapiro 1987] have described a way in which safe GHC can be compiled into Safe Concurrent Prolog which in turn can be compiled into Flat Concurrent Prolog or FCP. More recently Saraswat has proposed a way of translating GHC into a variant of Concurrent Prolog he calls CP($\downarrow,\mid$) [Saraswat 1987a]. Saraswat argues that a similar translation establishes that flat GHC/Parlog is a subset of FCP. Unlike the method of translation proposed here, Saraswat’s scheme of translation is not directed at the issue of implementing GHC. His mapping of GHC clauses to CP($\downarrow,\mid$) clauses so as to preserve success sets is too complex and unwieldy to be practical. Saraswat is only interested in arguing through his translation scheme that CP($\downarrow,\mid$) subsumes GHC in expressive power. However, in the absence of a serious implementation of CP($\downarrow,\mid$), this theoretical result lacks all practical significance.
Gregory observes in relation to Parlog about guard safety checks which are complex or impossible that

"It may be possible to find a way to combine the efficiency advantages of a compile-time safety check with the power of a run-time test where this is required." (p.132 [Gregory 1987])

The method of translation described in this chapter provides the means to augment a compile time safety check on a Parlog implementation with a run time suspension test.

4.5.6.1. Localisation

The central idea behind localisation [Taylor 1988] is that suspension effects due to the rule of synchronisation arise from what happens during guard evaluation and can be restricted to the locus of that evaluation. In what follows variables will be termed unsafe which occur both in the heads of GHC clauses and either in user defined guard literals or argument positions of system guard literals liable to bind given values. If all unsafe variables in guard literals are replaced by fresh variables then all transformed calls to guard literals can be allowed to proceed without being subject to a safety suspension condition. Each safety suspension condition can then be achieved by a special purpose system predicate which relinks the unsafe guard variable with its new replacement. In this way a GHC program can be distilled down to guarded Horn clauses interleaved with special primitives to achieve the safe guard suspension mechanism.

The function of the special primitive will be to allow the passing of values from the old variable to the new but to ensure that if unifying the new and the old guard variables would bind a calling argument variable before commitment, this special primitive will suspend. As an example take the GHC clause

\[
\text{head}([A]) \leftarrow \text{test}(A) \mid \text{true}.
\]

and transform it into Kernel GHC in the fashion established for Parlog [Gregory 1987] p.64-69. The transformation of GHC to Kernel GHC in order to make all input matching on head arguments explicit by being performed by extra literals in the guard is a likely part of a compilation strategy for GHC. Thus it is worth showing how a localised suspension test fits in with such a compilation strategy.

\[
\text{head}(A) \leftarrow [B] \leftarrow A, \text{test}(B) \mid \text{true}.
\]

Here \([B] \leftarrow A\) signifies unidirectional unification used for input argument matching. The left hand side is to be unified with the right, but only so that no variable on the right hand side is bound to a nonvariable. In such a case the unidirectional unification suspends until its arguments cannot be unified whereupon it fails,
or until it can unify them other than by binding right hand side variables to non-variables.

Replacing \( B \) by a fresh variable \( B1 \) in the user-defined guard \( test/1 \) and adding a special system predicate \( ward/2 \) to link the two variables in a way which protects the calling environment from being instantiated by suspending the call instead, gets the result:

\[
\text{head}(A) \leftarrow [B] \leq A, \ ward(B1,B), \ test(B1) \mid true.
\]

On general grounds it can be seen that \( ward/2 \) needs to be able to suspend on several variables because it must handle complex terms. Because complex terms can get progressively instantiated, it also needs to be able to suspend, awake and pass on values and then maybe suspend again. In other words it needs to be able to function transiently. Other properties of \( ward/2 \) can be inferred from the fact that the guard literal \( test/1 \) might need its argument to be instantiated to a value in order for it to succeed. The satisfaction of this guard can only be achieved if the \( ward/2 \) literal will pass bindings to its first argument from its second. However \( ward/2 \) literal should not allow a user defined guard literal to export a value from its satisfaction by having \( ward/2 \) simply unify \( B1 \) with \( B \). What is wanted is something more like unidirectional unification.

If \( ward/2 \) is equated with \( \leq/2 \)'s unidirectional unification, then the second goal of the guard will succeed in cases where it should not. The calling argument to which \( A \) is bound might get instantiated to \(_121\) and the first input matching unidirectional unification succeed sharing \( B \) with \(_121\). Since \( B1 \) and \( B \) are still uninstantiated, the second guard safety unidirectional unification can succeed by sharing them. Shortly afterwards the parallel \( test/1 \) call might bind the variable \( B1 \) to incorrect. The result could be the instantiation of a calling argument variable \(_121\) to incorrect before the clause has committed. This behaviour cannot be avoided by insisting that the unidirectional unification test be performed only after the \( test/1 \) literal has succeeded, because then bindings could never be passed through \( ward/2 \) into the user defined guard literal.

A better idea is to make the \( ward/2 \) predicate reluctant unidirectional unification so that the \( ward/2 \) predicate suspends if, in order to unify unidirectionally its arguments, it has to share variables. Reluctant unidirectional unification is unidirectional unification where bindings are made to the instantiable side whenever non-variable bindings are detected on the non-instantiable side and unidirectional unification is possible. It fails if the two sides can never be unified. Where instantiable side variables need to be associated with complex non-instantiable side terms containing variables, they are bound to a consistent copy of the complex
term to avoid variable sharing. The unidirectional unification is reluctant in that variables on the instantiable side only get instantiated to non-variables. They never get shared with variables on the non-instantiable side. The predicate suspends instead.

The problem now is that the test is too severe. If the guard literal needs a partially instantiated or uninstantiated argument in order to succeed, the \texttt{ward/2} predicate must not suspend for ever. For example if the guard literal was defined as follows:

\begin{verbatim}
  test(A) :- true | true.
\end{verbatim}

then the result of reluctant unidirectional unification would be for the \texttt{ward/2} predicate to suspend with two variable arguments even when the user-defined guard \texttt{test/1} has succeeded. What is wanted is for the \texttt{ward/2} predicate to suspend on two uninstantiated arguments but only for as long as the guard literal it is warding has not succeeded.

To implement the \textit{wait until the guard literal has succeeded} mechanism, a variable binding can signal that the user-defined guard literal has succeeded. The signal can be sent to the reluctant unidirectional unification predicate and the predicate can be defined in such a way that when it gets this signal, it no longer reluctantly unidirectionally unifies its arguments. Instead it tries to restore the linkage between the original variable in the guard literal and its replacement variable by non-reluctant unidirectional unification. This unidirectional unification will ensure that bindings generated by evaluating the guard literal are not passed out to the calling environment. It will cause the primitive to suspend should this be attempted. This unidirectional unification is stricter than the version of the primitive \texttt{\textless=2/2} Gregory uses for head argument matching, because of the need to ensure that calling argument variables on the right hand side do not become shared in the unidirectional unification process [Gregory 1987] p.64-69. In the mapping of Parlog to Kernel Parlog input matching on repeated head argument variables is performed separately from \texttt{\textless=2/2}, so no constraint of non-sharing of right hand side variables applies to Gregory’s version of \texttt{\textless=2/2}.

Putting these considerations together gives the following. The original clause

\begin{verbatim}
  head([A]) :- test(A) | true.
\end{verbatim}

gets transformed into:

\begin{verbatim}
  head(B) :- [A] \textless= B, ward(A1, A, Control), satisfy(test(A1), Control) | true.
\end{verbatim}
The first literal of the guard is merely to implement head argument matching. The next two literals are there to implement the original guard literal set about by the GHC suspension test. To do this another system predicate `satisfy/2` is introduced. It is a meta-call which attempts to satisfy its first argument. If it succeeds, it binds its second argument to some simple term and succeeds itself. If the attempted satisfaction fails, it fails. It is different from the Parlog meta-call `call/2` which never fails [Clark & Gregory 1984], although the meta-call `satisfy/2` can be defined using `call/2`.

The idea of the suspension mechanism is that up until the time that user-defined guard literal succeeds, the calling argument variable and its replacement are linked by a reluctant unidirectional unification which does not allow sharing of variables. If and when the guard literal succeeds, then the variable `Control` is instantiated by the metacall `satisfy/2`. This signals to `ward/3` not to suspend reluctantly the unidirectional unification. The relinkage with the original variable is established through this subsequent non-reluctant unidirectional unification which does allow sharing of variables between its two arguments.

### 4.5.6.2. General Localised Suspension

Transferring this translation scheme into a general scheme requires the taking into account of multiple occurrences of `unsafe` variables in one or more guard literals. The simplest sound way to do this is to place the entire original guard in a `satisfy/2` meta-call if any part of the guard contains an original head argument variable. An aggregate term of all unsafe variables and a consistent copy is also formed by substituting fresh variables for each unsafe variable, as the two terms related by `ward/3`.

The general rule for implementing guard safety suspension in GHC applies to guard literals containing user-defined predicates and system predicates liable to bind given values. After it has been applied, head argument matches can be decomposed into the guard in the fashion for transforming Parlog to Kernel Parlog.

1) The set of all variables occurring in a head argument and also occurring in the guard in a user-defined literal or in an argument of a system literal which is liable to be instantiated, is formed and called the `at risk` set. For each member of the `at risk` set a distinct fresh substitute variable is created. If there are no elements in the `at risk` set, nothing further is done.

2) Every occurrence of a member of the `at risk` set in a guard literal is replaced by its substitute.

3) The whole GHC guard is put into the first argument of a `satisfy/2` meta-call and a fresh variable to be called the `control` variable is put into its second argument.
4) A structure with an arbitrary functor and arguments consisting of all members of the at risk set of variables is formed. A copy of this structure is made but with substitute variables replacing at risk variables. A ward/3 predicate is formed with the structure copy as its first argument, with the structure as its second argument and with the control variable of satisfy/2 as its third argument. It is added to the guard as another literal.

4.5.6.3. A Localised Suspension Algorithm

Figure 4.1 gives a sequential algorithm for implementing ward/3. More efficient sequential algorithms exist and parallel ones probably exist as well. However, it is reasonably succinct and so conveys the basic idea simply. It returns one of three values each time it is executed - success, failure or suspend. It is executed once each time the ward/3 process is run until it returns success or failure. When it returns suspend, the process evaluating ward/3 should be hung on appropriate suspension lists for each relevant variable. Details concerning appropriate variables on which to suspend the ward/3 process have been omitted from the algorithm’s description but are not hard to determine. The algorithm presupposes that its arguments are able to unify. As lists can be represented by two argument structures, they are not handled separately. The algorithm uses a term catalogue for associating right hand side variables with left hand side variables. It is possible for a catalogued variable (sometimes called a catalogue term) to be bound by the Procedure part of the algorithm. Where copying is involved, uncatalogued right hand side variables get copied to fresh left hand side variables and the association gets catalogued. The whole Algorithm presupposes that it has exclusive write access to variables in the arguments of ward/3 during execution.
Algorithm

clear suspension flag and term catalogue
dereference third argument of \textit{ward/3}
apply Procedure to the first two arguments of \textit{ward/3}
IF Procedure returns \textit{failure}
THEN return \textit{failure}
ELSE IF third argument of \textit{ward/3} is not ground
THEN IF suspension flag is set OR catalogue is not empty
THEN return \textit{suspend}
ELSE return \textit{success}
ELSE IF suspension flag is set OR catalogue term is bound
THEN return \textit{suspend}
ELSE share variable pairs in catalogue and return \textit{success}

Procedure

dereference left hand argument \(L\) and right hand argument \(R\)
IF \(L\) is a variable
THEN IF \(R\) is a variable
THEN IF \(<L, R>\) is not a recorded pair
THEN IF \(<L, R_1>\) is a recorded pair
THEN bind \(L\) to \(L_1\)
IF \(<L, R_1>\) is a recorded pair
THEN set suspension flag
IF no \(<L, R_1>\) or \(<L_1, R>\) are recorded pairs
THEN record pair \(<L, R>\) in catalogue
ELSE IF \(R\) is a ground term
THEN bind \(L\) to \(R\)
ELSE copy \(R\) to new \(L_1\) in light of catalogued pairs adding
new variable pairs to catalogue and bind \(L\) to \(L_1\)
ELSE EITHER \(R\) is a variable
set suspension flag
OR \(R\) is a constant term
IF \(L\) is not same term
THEN return \textit{failure}
ELSE continue
OR \(R\) is a complex term
IF \(L\) does not have same functor and arity
THEN return \textit{failure}
ELSE FOR each argument \(L_i\) of \(L\)
apply Procedure to \(L_i\) and \(R_i\)
IF Procedure returns \textit{failure}
THEN return \textit{failure}
ELSE continue

Figure 4.1 Localised Suspension Algorithm
4.5.6.4. Adequacy of Localised Suspension

No variable, which occurs in the head of a GHC clause or in input matches <=/2, occurs in the guard literals which are put into the first argument of satisfy/2. Thus neither unifying head argument variables with the goal nor performing input matches will bind or share variables in the meta-call’s first argument goals. Since no other literal occurs in the clause’s guard apart from ward/3, only executing ward/3 can pass bindings into or out of the metacall’s goals. Since the metacalled goals are a consistent copy of the original GHC clause guard, their satisfaction conditions for the same input bindings must be the same. In particular, if the satisfaction of any original guard literal fails, then the meta-call satisfy/2 will fail. What remains to be argued is that ward/3 consistently passes input bindings, yet protects calling argument variables from being instantiated or from being shared with each other.

Before the meta-call completely satisfies the original guard literals whenever a right hand side term is discovered bound and its corresponding left hand side term is discovered unbound, the unbound term is bound to the bound term or to a copy of it. Since the first and second arguments of ward/3 correspond with the left and right hand terms of Procedure, this means that input values are always passed from the second argument of ward/3 to the first. The requirement for copying terms consistently ensures this is done consistently. The Procedure does not allow values to be passed from the left hand side to the right hand side and sets the suspension flag if any part of the left hand side is more instantiated than the right hand side. In this way the ward/3 predicate suspends if guard literals attempt to export values to the call’s environment.

Unidirectional unification in ward/3 can be transient because the only variables which can be instantiated are local to the guard. If ward/3 fails after previously making some bindings, no harm is done because the bindings are only local to the guard being warded. However, transient unidirectional unification can allow the unwanted communication of bindings if variables are first shared, and then a left hand side variable gets instantiated further. What makes this unproblematic here is that no variable sharing between the right hand side and the left hand side is allowed by the Procedure. The only place which allows sharing of right and left hand side variables is in the main part of the Algorithm after the meta-called guard literals have been satisfied and have signaled that this is so by instantiating the third argument of ward/3. Hence the value in the first argument of ward/3 is as fully instantiated as it will ever get from satisfying the guard literal. So there can be no sharing of variables between the first two arguments of ward/3, which the first argument then instantiates further and hence communicates unwanted bindings into the second argument.
4.6. Example Translations

An example shows the result of applying the transformation from GHC. The following GHC clause

```
process([A|B], C, [A|D]) :- check([A|B], E),
    test([A|D], E) | process(B, C, D).
```

is translated to the lingua franca clause

```
process(J, C, K) :- ([A|B], [A|D]) <= (J, K),
    wait((F, G, H), (A, B, D), I),
    satisfy((check([F|G], E), test([F|H], E)), I) | process(B, C, D).
```

The first guard literal implements the head argument match, the second implements the safe guard suspension test for all three relevant variables and the third satisfies the original guard. Only two extra system literals and a meta-call are required to implement the localised run-time suspension test and the head argument match. A different example shows the transformation of a Parlog clause

```
mode test(?, ?, ?).
    test([A|B], [A|C], pair(D, E)) :- check(A) &
        test(C) | ( (analyse(A), synthesise(B, F)) &
        (transform(B, F), unfold(C, E)) ) &
        test(B, C, D).
```

to the following lingua franca clause:

```
test(G, H, I) :- ([A|B], [A|C]) <= (G, H),
    wait(check(A), [], J),
    wait(test(C), J, K) |
    I = pair(D, E),
    wait(analyse(A), [], L),
    wait(synthesise(B, F), [], M),
    wait(transform(B, F), [L, M], N),
    wait(unfold(C, E), [L, M], O),
    wait(test(B, C, D), [L, M, N, O], P).
```
4.7. Translating back from the Lingua Franca

The reverse translation of translated GHC is simple. The general method presupposes all occurrences of the predicates \texttt{ward/3}, \texttt{<=/2} and \texttt{satisfy/2} are solely a product of the original correct translation.

1) The \texttt{satisfy/2} predicate is replaced by its first argument.
2) The first two arguments of \texttt{ward/3} are unified and the literal is removed.
3) The two arguments of \texttt{<=/2} are unified and the literal is removed.

If the guard is left empty, the space is filled by the primitive \texttt{true}. The simplicity of this reverse translation method is what it should be. All that is being done by translating GHC to the lingua franca is to decompose unifications in such a way that appropriate suspension effects and unification directionality can be associated with them. Clearly re-unifying them should restore the status quo.

Reverse translation of the lingua franca into Parlog is performed in three steps. First of all head argument matches are restored, then sequential conjunctions are put back and finally output head arguments are restored. If head argument matching has been performed in parallel with guard evaluation then it can be put back by unifying the two arguments of \texttt{<=/2}. Reverse translation of sequential conjunction elimination presupposes that it is only applied to clauses which have had their sequential conjunctions correctly removed.

The general method is applied separately to each distinct group of adjacent conjoined \texttt{wait/3} literals and generates the conjunction which replaces the original conjunction of \texttt{wait/3} literals.

1) Some conjunction operators between successive \texttt{wait/3} literals are replaced. If the prior \texttt{wait/3} literal’s third argument is a member of the posterior \texttt{wait/3} literal’s second argument, the sequential operator ‘&’ operator is substituted for the parallel operator ‘,’.

2) The following scope finding rule is recursively applied to the sequence of \texttt{wait/3} literals

   a) By proceeding right along the sequence of \texttt{wait/3} literals, the dominating parallel conjunction operator is sought. It is the first parallel conjunction operator such that no third argument variable of a \texttt{wait/3} literal on its left hand side occurs in the second argument list of a \texttt{wait/3} literal on its right hand side.

   b) If there is no dominating parallel operator, the dominating sequential operator is sought by searching right along the \texttt{wait/3} sequence. It is the first sequential conjunction operator where all third argument variables of \texttt{wait/3} literals on its left hand side occur as members of the second argument list in each \texttt{wait/3} literal on its right hand side.

   c) The arguments of the dominant conjunction operator are now determined by applying this scope finding rule to its left and right hand sides.
3) Once the scope of each conjunction operator has been recursively determined, each \texttt{wait/3} literal is removed and is replaced by its first argument.

Restoration of output head arguments is achieved by unifying the arguments of all \texttt{=/2} predicates in the body which were created by the original translation and removing them. The correctness of this reverse translation depends upon the original Parlog clause not using the same unification primitive. By allowing the same functionality to the user under a different name, this should present no problem. So long as standardised Parlog mode declarations are consistently used in Parlog clauses, where each argument \(A\) in clause head is input unless \(A\) appears in a unification '=' in body but not in the guard, then original Parlog clause mode declarations can always be correctly inferred from the initial translation to the lingua franca.

4.8. The Lingua Franca

The main characteristics of this CLP language can be summarised as follows:

- and-parallel conjunction only
- or-parallel and or-sequential search
- guards can be unsafe
- input matching is performed on all head arguments
- input matching precedes guard evaluation
- clause committal precedes clause body satisfaction

A BNF syntax used in a lingua franca implementation [Taylor 1988], which apes the Prolog standard syntax, is given for the lingua franca, GHC and Parlog in appendix A. The lingua franca differs from GHC in allowing sequential search and from Parlog in disallowing sequential conjunction operators. It differs from both in sequencing head argument matching before guard evaluation. This sequencing can always be fully compensated for, where the guard is not empty, by performing the input match using \(<=/2\) in the guard in parallel with the original guard goals. Its guards are potentially unsafe like Parlog’s guards. This enables it to translate both valid and invalid Parlog programs. Unlike Parlog its lack of guard safety is mitigated by its ability to support a run-time safety test anywhere safe guard evaluation is required. The simple properties of the transformation to achieve this can be supported by two way translation and so hidden from the user.

When used for programming in its own right, the lingua franca’s prime sequencing constructs are the commitment operator, sequential search and the \texttt{instream/1} input/output primitive. The lingua franca has not
been given a sequential conjunction operator in order to support a programming style which helps extract and-parallelism from applications. Unless Parlog is explicitly used on top of the lingua franca, a user will be less prone to lapse into the sequential Prolog style of programming which dogs many Parlog programs.

The specification of Parlog does not contain any proscriptions against particular kinds of primitives. Instead a fairly small set of low level primitives are recommended, and users are expected to define higher level predicates in terms of them [Gregory et al 1989], [Gregory 1987] p.147-154. On the other hand GHC explicitly proscribes time of call primitives like \textit{var/1}, primitives violating anti-substitutability, and input or output primitives other than \textit{instream/1}. Although a similar sort of stream handling predicate can be defined in the lingua franca or Parlog in terms of input and output Prolog-like primitives like \textit{get/1} and \textit{write/1}, GHC disallows direct use of such primitives. Thus this strategy of translating away such primitives cannot be adopted by the lingua franca unless the translation of GHC clauses to the lingua franca’s clauses was allowed to abandon a one-to-one policy of translation. The obvious solution is just to admit \textit{instream/1} as an extra primitive in the lingua franca. All standard GHC and Parlog primitives can be supported by the lingua franca. It should also support the primitives needed for translating GHC and Parlog to the lingua franca \textit{\leq/2}, \textit{satisfy/2}, \textit{wait/3} and \textit{ward/3} and any others the lingua franca might require on its own account.

Translations of GHC and Parlog clauses need only use a subset of all the lingua franca primitives.

Although \textit{ward/3} can be used to compile GHC to the lingua franca, it cannot be used with \textit{satisfy/2} to compile GHC to flat GHC. While the meta-call \textit{satisfy/2} does not violate anti-substitutability unlike the Parlog meta-call \textit{call/2} [Ueda 1986a]4.7.2, the primitive \textit{ward/3} does. This prevents it from being made a primitive of GHC or any subset of GHC.

When viewed as an intermediate language in the compilation of Parlog, the lingua franca plays a similar role to Parlog to that played by Kernel Parlog [Gregory 1987] p.64-69. However, there are several important differences between the lingua franca and Kernel Parlog. Unlike Kernel Parlog the lingua franca is intended to be suitable as a programming language of first instance like Parlog and GHC. Thus it preserves the head argument matching functionality of GHC and Parlog rather than translating it away in the fashion of Kernel Parlog. Furthermore Kernel Parlog does not possess primitives like the lingua franca’s \textit{ward/3} primitive which enable the lingua franca to translate away the GHC run-time suspension test. The complex action of a primitive like \textit{ward/3} cannot be performed by the simple input matching primitives of Kernel Parlog alone. In order to exhibit the behaviour of \textit{ward/3} the action of such input matching primitives would
have to be organised so that several of them could be executed together atomically. However, Kernel Parlog lacks the means to do this. Saraswat has described part of a plausible looking method for translating away the run-time suspension test [Saraswat 1987a] into a whole series of simple matching actions performed by primitives in parallel in the context of translating full GHC into the CLP language CP[↓, |]. However, Saraswat is able to exploit the multiple environment property of this CLP language where atomic operations of the required kind are possible. Kernel Parlog cannot follow him in doing this. Nor is the primitive \texttt{ward/3} suitable for adding to the primitive set of Kernel Parlog. Part of the design philosophy of the primitives of Kernel Parlog is that a primitive never requires to suspend on more than one variable at once. Indeed Kernel Parlog has the elegant property of having only one suspending primitive \texttt{data/1}, which suspends on its sole argument, if it is an unbound variable. Unfortunately, it is essential to the operation of the primitive \texttt{ward/3} that it be able to suspend on multiple variables. With $A$, $B$ and $C$ unbound, $\texttt{ward((1,2), (A,B), C)}$ must suspend on both $A$ and $B$ to behave correctly.

4.9. Conclusion

A strategy for absorbing the differences between two prominent CLP languages by assimilating them into a common language variant has been proposed. It meets different demands of concurrent logic programming languages by supporting each in a common medium. It is a strategy which might be extended further to encompass other desirable features of CLP languages like atomic output committal in the fashion of GDC [Burt & Ringwood 1988, Ringwood 1988]. The strategy has detailed how to translate both GHC and Parlog into a lingua franca and back again so that an implementation of the lingua franca can be transparently presented to a user as either executing GHC or Parlog directly. By supporting both languages the lingua franca can provide a user with the semantically desirable properties of GHC, the control features of Parlog and any performance benefits its simplicity might bring on a single implementation.
Chapter 5

5. Aspects of the Design of a CLP Engine

5.1. Introduction

This chapter considers aspects of the design of an emulator for supporting the lingua franca of GHC and Parlog described in the last chapter. Its focus is on implementation issues local to one processor, but the discussion is intended to be relevant to shared memory multi-processor implementations of non-flat CLP languages as well. An execution model for the lingua franca is proposed along with a memory management scheme for realising the execution model on one processor. It has many points in common with two implementations of Parlog, SPM [Gregory et al 1989] and the JAM [Crammond 1988]. A distinctive feature is that it allocates memory for storing logical variables from temporary goal argument storage and not from the heap. This significantly reduces heap usage and avoids having to garbage collect these memory cells. Schemes for handling scheduling, meta-calls, sequential and parallel clause search, input-output and multiple variable process suspension are also described. Differences between these schemes and those employed in SPM and the JAM are summarised in the conclusion. These schemes are intended to contribute to expanding the space of implementation techniques for realising the CLP engine. Descriptions of these schemes follow the terms of a uni-processor implementation of the lingua franca which was developed in C as an interpreter [Taylor 1987].

5.2. A CLP Execution Model

Execution of the lingua franca is by and-parallel input resolution. It can be realised by a variant of the AND/OR execution model for committed choice execution of non-flat guarded clauses described by [Takeuchi & Furukawa 1986] p.246 and by [Gregory 1987] chapter 6, which has been employed in the SPM implementation of Parlog for uni-processors [Gregory et al 1989], and the JAM implementation of Parlog for shared memory multi-processors [Crammond 1988]. This model is itself an adaptation of the original AND/OR tree execution model for SLD resolution over definite clauses [Kowalski 1979, Conery 1986]. An alternative execution model for committed choice languages is based on AND-trees [Gregory 1987] chapter 7. It can be used for shared memory parallel execution schemes for non-flat CLP languages.
[Levy 1986a]. Its more coarse grained process structure, where all the guards of a relation are examined in turn in a single processing step, makes it more suitable for implementations of flat CLP languages like the FPM implementation of flat Parlog [Foster & Taylor 1987], and the PIM/p implementation of KL1 (flat GHC) [Sato et al 1987]. It is not so suitable for the deep guard committed choice resolution required by a coupled resolution architecture.

A variant of the AND/OR tree model for committed choice resolution called the Goal-Clause execution model is sketched. It provides a straight-forward way of supporting and-parallel input resolution. Its main theme is that a call begins as a goal process which spawns clause processes to find a suitable clause to reduce the call. Goal processes coordinate or-parallel clause search before the call is reduced. Clause processes control the joint satisfaction of the goals in the guard and body of a clause. Child processes in turn spawn more children forming a tree of processes. At the leaves of the tree primitive goal processes or facts end the growth of that branch. The tree grows by spawning and contracts when processes succeed, fail, or when a goal process is replaced upon the goal’s commitment to a clause by goal processes representing the body goals of that clause. Significant differences from a related execution model [Crammond 1988] chapter 3, include allowing delayed clause process promotion for a logical variable storage optimisation, and giving goal processes a lazy clause process creation role.

5.2.1. Goal-Clause Execution Model

The Goal-Clause execution model is centred on a tree of processes of two main kinds

- goal process  OR-like
- clause process  AND-like

Goal processes are OR-like, because they have child clause processes representing clause alternatives for reducing the goal. Committed choice resolution means that only one of these children succeeds. Clause processes are AND-like processes, because they have child goal processes representing conjuncts of the guard and body of the clause. All of these children must succeed for the clause process to succeed. Goal processes can be either

- literal  user-defined goal
- primitive  built in predicate goal
Literal goal processes represent goals defined by user supplied clauses, and primitive goal processes represent calls to built in predicates. Clause processes are

- untrusted goal has not committed
- trusted goal has committed to clause

Untrusted clause processes spawn children to evaluate the guard of a clause, and become \textit{trusted} when that clause commits. Trusted clause processes are not always immediately removed from the execution tree for reasons discussed below. Siblings of processes may be or-parallel or and-parallel processes. Untrusted clause processes have or-parallel siblings, while goal processes and trusted clause processes have and-parallel siblings. Unlike the more heterogeneous SPM model [Gregory et al 1989] there are not parallel and sequential versions of OR-like and AND-like processes. The lingua franca lacks a sequential conjunction operator so the model only needs a single parallel AND-like process. Furthermore sequential search is achieved by constraining clause process creation rather than by a special sequential OR-like process type. Thus only two main process types suffice.

5.2.2. Process States

Processes in the Goal-Clause execution model can be in one of six states

- executing being executed
- meta-suspended suspended by its meta-parent
- runnable waiting to be executed
- suspended on child suspended on one or more children
- suspended on input suspended waiting on input on a stream
- suspended on variables suspended on one or more variables

Processes are initially created as runnable and are added to the runnable process ring (in the prototype’s case a doubly linked ring of processes which represents the queue of processes waiting to be run). When they are executed, their status changes to executing. Processes which succeed or fail are immediately removed from the execution tree. Each process bar the top level goals also has a meta-parent. This is a primitive goal process, representing a control meta-call somewhat similar to SPM’s [Gregory et al 1989]. Should the meta-parent be required to suspend execution of its sub-tree, each process in that tree is meta-
suspended. Should a process spawn children, it suspends waiting upon its children to resolve their execution status so that it can succeed or fail. Processes can also be suspended upon an input stream, which is either a terminal, a pipe or an ordinary file. They can also be suspended upon one or more variables. The ability to suspend a process upon multiple variables is a significant feature of CLP implementations which is not possessed by some Parlog implementations like SPM [Gregory et al 1989] and by early and now superceded versions of the JAM [Crammond 1988]. However, it is exhibited by flat CLP implementations like the FPM [Foster & Taylor 1987], and the PIM/p [Sato et al 1987]. The ability can be used to confer significant expressive power on primitives which are executed as a single process (like ward/3 discussed in chapter 4), and it is important for handling clause head matching efficiently.

5.2.3. Roles of Process Types

The role of a primitive goal process is to:

- represent a built in predicate prior to attempted reduction
- represent suspended built in predicate goals

Most primitive goal processes are leaf processes which simply succeed, fail or suspend. However, a few primitive goal processes are meta-calls which create offspring and wait upon all of their offspring to succeed or for one to fail. Most primitive goal processes are satisfied or fail in a single step. The remainder suspend. There are several ways for a primitive goal process to suspend. It can be suspended

- upon argument variables
- upon its children
- upon an input stream
- by its meta-parent

A primitive goal process can suspend upon one or more argument variables waiting for them to be instantiated or shared with another variable. It can also suspend upon its children in the case of meta-calls, waiting for all to succeed or one to fail. Thirdly, a primitive goal process can suspend upon an input stream in an ordered queue. A primitive goal process can also be suspended by its meta-parent. Suspension upon a Prolog process (e.g. the state of a call to a Prolog relation in the CLP engine which is awaiting the result of the Prolog computation) can be realised among other ways by suspending the call to the Prolog process upon
input to the CLP computation along their communication channel.

The role of a literal goal process is to:

- represent user-defined guard or body goals prior to attempted reduction
- control clause process spawning in the search for a clause to reduce goal

Thus the goal process represents the evaluation of a goal up until the goal is reduced to the body of a particular clause. It does this by spawning clause processes to evaluate each (relevant) clause for the relation in the goal call. When one of its child clause processes trusts, all of that clause process’s or-parallel rival clause processes are killed off. The children of the clause process are then promoted in the place of the goal process, or if delayed promotion is used, the sole remaining clause process is promoted in place of the original goal process. Its new siblings are and-parallel goals.

The role of a clause process is to:

- conjoin the satisfaction of a clause’s guard and body goals
- initiate head argument matching and spawning of guard and then body goals on trusting
- maintain links with siblings and offspring for location during termination

When a clause process is created and run, the goal is matched with the clause head. If this succeeds, processes are spawned to evaluate any guard goals, and the clause process suspends upon its children. If there are no guard goals, the clause process tries to commit (trust) immediately. If head argument matching signals the need for the goal to be instantiated further, the clause process is suspended upon these variables. Otherwise the clause process fails. Thus a clause process can be suspended upon goal argument variables, upon its children, or by its meta-parent.

When an untrusted clause process suspended upon one or more variables is made runnable again upon one of these variables being shared or instantiated, the process re-attempts head argument matching upon being run. When an untrusted clause process’s last child succeeds, the clause process is scheduled and run. If the child of an untrusted clause process fails, the clause process itself fails. An important optimisation here is to avoid spawning guard goal processes where only simple primitive goals occur in the guard (their role can be performed by special purpose instructions in a compiled implementation’s instruction set), and to suspend the clause process on all variables in the simple guard which require further instantiation. If all child guard goals succeed, the clause process tries to commit in the fashion explained below.
5.2.4. Lazy Clause Process Creation

Under a lazy clause process creation scheme, goal processes do not create clause processes for each clause for the relation which has to be searched in parallel as happens in the JAM [Crammond 1988] chapter 3. Instead clauses can be spawned lazily as happens in the SPM [Gregory et al 1989] p.410. On a uni-processor implementation this means one at a time. On a multi-processor implementation how lazy clause process creation should be, can be made to depend upon how loaded other processors are. Thus if general loading is heavy (i.e. runnable process rings are large) one clause can be created and scheduled each run. If loading is light on other machines, several clause processes can be created and scheduled locally at the same time. They can then be picked up by other processors should they seek work - i.e. by on-demand load balancing [Sato and Goto 1988]. There are several optimisations here. The goal process can use indexes created at compile or consult time on clause head arguments to avoid spawning clause processes for clauses which could never head match the goal [Crammond 1988] 3.1.4. Indexing can also establish that only one clause head can match the goal. This makes spawning a clause process unnecessary, as the goal process can execute the guard and then the body goals directly [Crammond 1988] 3.1.4. Lastly where only head matching is required and the clause has no guard or body goals, then the goal process can succeed or fail at one go.

When the goal process spawns a clause process, it is scheduled after the clause process. Its next clause offspring is not spawned until it is run again. This carries on until the next clause, which can be searched, is required to wait until all prior clause guard evaluations have failed, or until all clauses have had clause processes spawned to evaluate their guards. Then the goal process suspends upon its clause children. Should its last child clause process fail, then the goal process can be woken up again. It may need to create and schedule further clause processes over which the failing clause process had search priority. A sequential search operator gives all clauses preceding it search priority over all subsequent ones.

When a clause process fails, a check is made with its literal goal parent to see if the next uncreated clause process would have the same search priority as prior clause processes. If it would have or if no other clause process exists, the next clause process is immediately created and scheduled next. If no other clause process can be created to meet this demand, the goal parent of the clause process is failed as well. The result is to achieve a lazy creation and scheduling policy for clause search.
Figure 5.1 Lazy Creation of Clause Processes

The following lingua franca programs

\[
\begin{align*}
\text{is}(X, Y) & : \text{ask}(X) \quad \mid \text{are}(X, Y). \\
\text{is}(X, Y) & : \text{if}(X), \ \text{else}(Y) \quad \mid \text{were}(X, Y).
\end{align*}
\]

\[
\begin{align*}
\text{in}(X) & : X == \text{on} \quad \mid \text{above}(X). \\
\text{in}(X) & : X == \text{up} \quad \mid \text{over}(X).
\end{align*}
\]
can be used to illustrate lazy clause process creation during the execution process. If the goal \( is(A,B) \) is being executed then figure 5.1a shows the state where a clause process has been created to search the first clause. The clause process spawns a goal process \( ask(A) \) to evaluate the first clause’s guard. This in turn may involve spawning more clause and goal processes. Should this continue sufficiently long then the still active goal process may get rerun and spawn another clause process to evaluate the second clause, as is illustrated by figure 5.1b. Figure 5.1c shows the evaluation state of the goal \( in(up) \) where a clause process to evaluate the first clause for \( in/1 \) has been spawned. The failure of the primitive goal process \( up == on \) causes its parent clause process to fail. This results in the emulator spawning another clause process to replace it as is shown by figure 5.1d. If the clauses for \( in/1 \) had contained a sequential search operator as follows,

\[
\begin{align*}
\text{in}(X) & : \ X == \text{on} \mid \text{above}(X); \\
\text{in}(X) & : \ X == \text{up} \mid \text{over}(X).
\end{align*}
\]

then only the failure of all the clause processes searching clauses before the sequential search operator would be allowed to initiate the creation of clause processes to search clauses after the operator. This is straightforward to implement by keeping state information in the goal process. This scheme integrates the mechanism for handling lazy clause process creation with the method for realising sequential clause search.

For clauses without very deep guards lazy creation has the advantage of economising on process spawning. This should reduce the number of process descriptors allocated and thus economise on active memory usage. It should also markedly reduce the number of clause processes which are created and killed off without ever having had a chance to be executed because an earlier clause process commits before this happens.

### 5.2.5. Process Promotion

When the child clause process of a goal process commits because head matching requirements and guard goals all succeed for that clause, then the goal process can be replaced by the goal processes representing the goals in the body of that clause. This process can be illustrated for the goal \( test(7) \) for the program

\[
\begin{align*}
test(A) & : \ \text{odd}(A), \ A > 5 \mid \text{up}(A, B), \ \text{try}(B). \\
test(A) & : \ \text{even}(A) \mid \text{filter}(A, \text{prime}), \ \text{revise}(A).
\end{align*}
\]
using clause-goal process diagrams. Figure 5.2a shows two clause processes existing for the goal test(7). Each is competing to reduce the goal. Each has head matched the goal arguments with its clause’s head arguments and has spawned goal processes to evaluate the guard to see whether the clause should be selected for commitment. Figure 5.2b represents the state where the guard goals for the first clause have succeeded, but the guard goal for the second clause has still not been run.
Figure 5.2 Committed Choice Reduction of a Goal
If the first clause process is run, it will recognise that its guard children have succeeded, commit by killing off its rival clause, and spawn two goal processes to evaluate its body. This is represented by figure 5.2c. Figure 5.2e shows the two new processes which have been spawned to evaluate the body of the first clause. However, they have been promoted in place of their grandparent goal process and their parent has been removed. This represents the normal optimisation. Memory management of goal process argument storage suggested the alternative delayed promotion option of figure 5.2d. In it the intermediate clause process is promoted on top of the original goal process where there is more than one body goal, and the untrusted clause process is turned into a trusted clause process. Promotion of the body goals is delayed until only one is left. It is then promoted on top of its trusted clause process parent. Normal body goal process promotion is still used where there is only one body goal process. The reason for delaying promotion is to economise on using the heap during goal argument storage.

Figure 5.3a shows a more detailed representation of the Goal-Clause diagram of Figure 5.2c. The extra boxes depict argument records which store the argument values of goals. Arrows between these long boxes represent variable pointers. Argument records are of certain fixed lengths like SPM’s argument vectors [Gregory et al 1989]. The variable B, which is shared between body goals up(7,B) and try(B), is stored in the next unused cell on its grandparent goal process’s argument record. The argument records for the two body goal processes both contain variable pointers to this cell. This method of storage allocation for a logical variable departs from the normal practice of allocating storage for such variables on the heap to avoid the risk of creating dangling pointers. This risk arises because it is unknown at the time of the creation of the body goal processes which body goal process will finish executing first. Thus the storage space for the shared variable cannot be allocated in the temporary storage area normally assigned to the arguments of either process in case that space gets deallocated when that process finishes executing, but before the other process is finished with that variable. However, the normal approach suffers the disadvantage of rather significantly increasing consumption of semi-permanent memory for all these logical variables. This extra memory consumed can only be recovered by time consuming garbage collection.
Figure 5.3a

Figure 5.3b

Figure 5.3  Delayed Promotion Scheme
The approach advocated here is to store variables shared between guard and/or body goals, which do not occur in the clause head, in the argument record of the parent goal process rather than in the heap. This is done whenever there are unused cells in the grandparent goal’s argument record. Otherwise heap cells can be allocated for these variables. A parent process waits on its children finishing execution before finishing execution itself. Hence using a memory cell in the grandparent goal’s argument record promises safe storage for such a variable for as long as either of the two offspring goal processes sharing a variable need. However, process promotion to recover space makes the story rather more complex. Figure 5.2c represents the unoptimised situation without process promotion. The normal optimisation is shown by figure 5.2e. However, the attempt to apply the optimisation to the amplified form of figure 5.2c showing argument records, figure 5.3a, runs into a difficulty. If the bottom half of figure 5.3a is promoted over the top half, so that the top goal process, its child clause process and the top goal process’s argument record are recycled, then the pointers to the logical variable’s memory cell become dangling.

To rectify this situation the delayed promotion scheme illustrated by figure 5.2d and amplified by figure 5.3b is applied instead. The clause process for test/1 is promoted on top of its parent goal process. It now becomes a trusted clause process and inherits its parent’s argument record. This trusted clause process persists so long as it has more than one goal process offspring. It serves as a place holder in the process tree which keeps its associated argument record active. When the trusted clause process has its second last child process succeed, the remaining child can be promoted on top of the trusted clause process and the process descriptor and its argument record can be recycled. This recycling can be made safe from creating dangling pointers by checking that the child process’s argument record does not have pointers which point to cells in its parent’s argument record. If that happens, the pointers can be either short circuited one level higher in the process tree or made to point at a new cell on the heap. To make sure that promotion is not compromised by variable pointers in argument records which leap levels in the process tree and point into the argument records of ancestor processes, variable management is subject to the following rules.

- all lists and structures are stored on the heap
- heap pointers never point out of the heap
- binding a variable to a constant is performed by copying the tag and content of the constant onto the tag and content of the variable
- variables shared during explicit unification by unification primitives point at a newly
allocated cell on the heap

• variables shared in head argument matching point at the first pointer in a chain of pointers on argument records or directly at a cell on the heap

The first three rules are standard practice on Prolog implementations [Warren 1983]. The fourth rule is rarely applied, but is necessary to ensure that pointer chains always point at the heap or up the process tree. The last rule ensures that variable pointers from argument records to other argument records point only to the level immediately higher in the process tree. This localises the need, when promoting a trusted clause process’s last child on top of the clause process, to investigate any argument records other than the trusted clause process’s argument record and the child’s argument record.

A minor disadvantage of this scheme is that it delays recycling process argument descriptors and argument records by delaying promotion. However, the delay does not abandon the crucial tail recursion optimisation of promoting child goals to the generation of their parent in order to keep the process tree from growing ever longer (c.f. the tail forking optimisation of SPM [Gregory et al 1989] p.395). Promotion is only delayed for as long as a parent has two or more children, and then performed as soon as one or the other succeeds or fails. The number of such delayed promotions is not expected to be frequent relative to the number of processes. In theory certain cases like \( g/1 \)

\[
g(X) :- \text{true} \mid g(Y), g(Y).
\]

would suggest the creation of \( 2^n - 1 \) extra trusted clause processes for each \( n \) levels of delayed promotion, but in practice a depth first process creation and scheduling policy (even if bounded) works against the exponential creation of trusted clause processes and tends to hasten promotion by reducing child processes to the one that may be promoted on its trusted clause parent. Furthermore, \( g/1 \) is an artificial example. Real programs do not recurse for ever on multiple goals because their authors want to avoid exponential process growth. Where real programs recurse for ever, they do so on single goals where the delayed promotion scheme can be applied to keep the process tree from growing ever larger. If the scheme has a significant drawback, it lies in its prevention of the short-circuiting of variable pointer chains running through temporary argument storage (i.e. from argument record to argument record), although those that end on the heap can be shortened. This is likely to increase dereferencing time significantly where the process tree grows deep. This has to be set against the merits of rather significantly decreasing the rate of consumption of memory which has to be garbage collected.
5.3. Memory Management Issues

Mapping this execution model onto an abstract implementation architecture requires a memory management scheme. Memory is a key resource. It has to be managed economically to keep peak usage within sustainable limits and to ensure that it is effectively recycled at low cost. Shared memory multi-processor implementations must cope with contention among processors in accessing and updating shared memory, and must cope with load balancing and inter-processor communications. This creates special requirements not faced by uni-processor implementations. Nevertheless, most considerations governing memory management of an implementation of this execution model are similar to those faced on uni-processors.

5.3.1. Representing Processes

The key data structure so far discussed is the process. It needs to be described by the following fields.

- **Type**
  - goal (literal or primitive) or clause (untrusted or trusted)
- **Code**
  - clause code or built in predicate identifier
- **Arguments**
  - goal’s calling arguments
- **Meta-parent**
  - meta-parent of process
- **Relatives**
  - parent, siblings, children
- **Offspring**
  - count of children
- **Group**
  - active or suspended process group
- **Status**
  - execution state, resources

Similar fields are used by other CLP implementations like the SPM implementation of Parlog [Gregory et al 1989]. Processes belong to a group of processes. They either belong to one of the following:

- ring of runnable processes
- ring of processes suspended upon a variable
- queue of processes suspended upon an input stream

or are suspended alone on their children. Clause and primitive goal processes can suspend upon one or more variables, certain primitive processes can suspend upon an input stream, and clause, goal and meta-call primitive goal processes can suspend upon their children. Processes, which can be suspended in more than one way, only need to be suspended in one way at a time. The status of a process records its execution
state, and the resources it has left to use.

In execution models such as used in the JAM [Crammond 1988] chapter 3 trusted clause processes never exist, because clause processes cease to exist when their clause commits. However, the delayed process promotion scheme described earlier requires this distinction. Following the approach of [Gregory et al 1989] processes are linked to their parent, their siblings and their children. This contrasts with the approach of [Nystrom 1988] following the earlier paper of [Crammond 1986] where AND and OR-like processes only record pointers to their parents and record no pointers to their children or siblings. This is an interesting optimisation, which significantly saves on pointer storage and makes process relocation more painless. However, it introduces awkward complications to process control management. When a clause process commits, rival clause processes can no longer commit. Thus they and their offspring can no longer contribute to a solution of the goal. However, in the absence of pointers to these siblings it is tricky to locate these siblings. In the absence of pointers from these processes to their children, it is impossible to locate them or their offspring. Thus none of them can be terminated early. As a result, orphaned child processes of guards, where a different clause has committed, may carry on chewing up computing resources doing no useful work. This is recognised by Crammond, who proposes an indirect kill scheme [Crammond 1988] 3.6.1 and 3.7, whereby a special pointer is passed to offspring processes so that offspring can recognise whether they should kill themselves off.

Offspring processes also need to be located for other purposes. Adequate systems programming in CLP languages [Foster 1988] requires means like Parlog’s stream controlled meta-call [Gregory et al 1989] to manage sub-computations adequately. Among other things this control meta-call must be able to suspend execution of all processes in its subtree. In the absence of links from parents to children this is impossible for the parent to organise. Crammond proposes an extension of his indirect kill scheme whereby processes at suitable points in their execution use their root pointer to point to a chain of processes including their meta-parent to determine whether they should kill themselves off or meta-suspend themselves. At this juncture Crammond’s scheme becomes strained and less than convincing. Although as Crammond points out, using pointers to siblings and children would consume more memory, their use allows the computation tree to be recorded in a straight-forward fashion, which is important in retaining simplicity in the implementation scheme. More importantly their use saves time in locating offspring.
5.3.1.1. Meta-call Process

All processes bar the top level meta-call have a meta-call parent. Meta-call processes behave somewhat like clause processes in AND-ing together the results of their offspring. The top meta-parent is given a virtual implementation. It always exists during evaluation of a query and the success or failure of its evaluation determines the result of the query. It is not stored in the process tree but has its status represented by a set of status registers which record

- active process
- meta-quota
- meta-resources
- runnable processes
- input suspended processes
- suspended processes
- Prolog Unix process

All meta-calls have a meta-call status block which records these values for the meta-call’s computation tree. When a meta-call starts or resumes execution, these values are loaded into state registers. They are saved in the status block when the context of a meta-call is switched out. The active process register records a pointer to the currently active process. This will either be null if there are no active processes, or it will point to one of the processes in the runnable process ring. The meta-resources register records the number of execution cycles allowed to the meta-call computation before it is rescheduled to the last position in its meta-parent’s runnable process ring. During execution of processes, which are directly within this meta-call environment, this register is decremented by one for each cycle of execution. When the parameter becomes zero, a new quota of resources is given to the register before the meta-environment context switch is initiated. The runnable, input suspended and suspended process registers record the count of these processes directly within the meta-call environment. The contents of a meta-call environment are hidden from encompassing meta-environments. The Prolog process identifier is initially null, but becomes the Unix operating system’s process identifier should a Prolog process be created by calls to Prolog in that meta-call’s environment.

Reference to a meta-parent process is indicated by a null process pointer. Evaluation of a query consists in creating one child process for each query goal and evaluating them. If any fail then the meta-parent process
fails immediately. If any succeed then the count of top level processes is decremented by one. When the count reaches zero, the meta-parent process has succeeded and thus the query has succeeded. By zeroing the count of top level query goals, execution can be terminated gracefully after a user interrupt. Resumption of execution is achieved by resetting the register to its former value and recommencing the execution cycle where it left off.

5.3.1.2. Goal Process

Some child processes are created by their parent clause process when it is created for the first time. In a successfully executing clause one goal process is eventually created for each user defined goal in the clause code, and one primitive goal process is eventually created for each built in predicate. Compiled implementations can make much of this primitive goal process creation virtual by directly executing relevant parts of the target machine language’s instruction set instead. Along with each process descriptor, a data record is created to record the current values of the arguments, if any, of that predicate. Processes to evaluate guard goals are created and scheduled immediately, whereas processes to evaluate body goals are created and scheduled only when their parent clause process trusts. Besides normal information a goal process records information on

- clause code for the goal
- next unscheduled clause
- constrained search flag

The clause code pointer indicates the relevant part of the clause code. Information on the next unscheduled clause is also recorded. A null field signifies that all clauses for that relation have been scheduled. The information reveals whether the scheduling of the next clause should be unconstrained or constrained only to happen when all existing clauses have failed. These options correspond to parallel or sequential clause search. The process records details on children clause processes spawned to find the clause which may reduce the goal call. The goal process lasts only until the call it represents trusts and then its sole child trusted clause process is promoted on top of it. The goal process is also linked to all fellow conjuncts of its clause’s guard or body, and into the runnable process ring. When goal processes are run, they either create and schedule guard goal processes and a clause process to evaluate the goal or suspend upon their children if no more clause processes require to be created. If head argument matching for the first clause succeeds
immediately and there are no guard goals, the goal process trusts immediately.

A primitive goal process is often satisfied or fails immediately on being run. Other primitive goal processes can suspend waiting for one or more variables in arguments to become further instantiated. Appendix B contains a complete description of the user’s view of the primitives supported by the prototype implementation of the lingua franca. Primitive goal processes, if they are meta-calls, can also suspend upon their children. In the case of the stream controlled meta-call call/3 the primitive goal process does not suspend upon its children. It waits for a limited number of cycles and is rescheduled to the end of its active process ring to allow other processes a fair chance of being executed, if that number of cycles has not been sufficient to finish executing its child goals. Certain input primitives may also suspend upon files which are open for reading. They are woken and run, if input to meet their execution conditions arrives on that file. If evaluation of the call/3 meta-call is completely stalled waiting upon input, the meta-call itself goes to sleep waiting for any processes in its evaluation tree to be woken up by fresh input.

Besides normal information a primitive goal process records

- identifier of built in predicate
- children in the case of meta-calls

When a primitive goal process is woken up again, it is given a full quota of new resources. When it is suspended on one or more variables, the primitive goal process records the number and addresses of suspension ring attachments of the process using data records. Details are given later relating to figure 5.4. If the primitive goal process is suspended upon an input stream, it records its meta-parent and its relation to other primitive goal processes suspended upon that stream. If it is suspended upon its children, it records the current number and addresses of child processes owned by the process. Four out of the five meta-calls \\
+
, call/2, satisfy/2, and wait/3 may employ it in this fashion.

Two meta-call built in predicates satisfy/2 and wait/3 have their single goal spawned as a child process at the same time that the meta-call is spawned as a primitive goal process. The child is scheduled first and the meta-call is scheduled after it. On being run the meta-call checks whether its child has succeeded or failed and suspends upon its child if it has not. By contrast another three meta-call built in predicates call/2, call/3 and \\
+
 get executed by dynamically creating one or more child processes to evaluate their goal argument on being run with arguments which are sufficiently instantiated. The meta-call primitives \\
+
, call/2 and
wait/3 can suspend upon variables or suspend upon a child but never require to do both at the same time. The meta-call primitive call/3 can suspend on a variable or upon relevant input streams but never requires to do both at the same time.

The following four special built in predicates are provided to emulate GHC and Parlog over and above multi-modeled equivalents of standard Prolog built in predicates.

\[
\begin{align*}
A & \leftarrow B & \text{instantiate } A \text{ only in order to unify } A \text{ with } B \\
satisfy(G,S) & \text{ satisfy } G \text{ and ground } S \text{ if } G \text{ succeeds or fail with } G \\
wait(G,L,S) & \text{ if } L \text{ is ground, satisfy } G \text{ and ground } S \text{ if } G \text{ succeeds} \\
ward(X,Y,S) & \text{ if } S \text{ bound one-way unify } X \text{ & } Y \text{ else unify } X \text{ & fresh copy of } Y
\end{align*}
\]

\(\leftarrow\)/2 is a one way unification predicate which will only instantiate its left hand argument in the attempt to unify it with its right hand argument. The predicate functions transiently, and will suspend if in order to unify its arguments, the right hand argument would have to be instantiated.

satisfy/2 is a meta-call which is scheduled at the same time as its first argument children goals but after them. It waits upon the process or processes evaluating its first argument all succeeding or upon one of them failing. When all such processes succeed or one fails to be satisfied, it binds its second argument to an atom, if it has succeeded, or fails itself if a first argument goal has failed. satisfy/2 is used in conjunction with ward/3 to implement the GHC run-time suspension test.

wait/3 suspends on the list of variables in its second argument until it is a complete list of instantiated elements. It then meta-calls its first argument and instantiates its third argument to an atom if the call is satisfied or fails itself if the call fails. It only suspends upon one of the variables in the list in its second argument. wait/3 is used to sequence the satisfaction of goals in the lingua franca which in Parlog are conjoined by the sequential conjunction operator.

ward/3 performs transient reluctant one-way unification between its first two arguments, while its third argument is an uninstantiated variable. This may involve copying across complex terms with consistent variable replacements from the non-instantiable side to the instantiable side. In this mode the second argument is protected from having any of its variables instantiated or shared. The process evaluating ward/3 suspends if unifying ward/3's first two arguments would violate this condition. If the variable in its third argument becomes instantiated then it performs non-reluctant one-way unification between its first two
arguments. This one-way unification suspends if it can only succeed by sharing non-instantiable side variables. \texttt{ward3} fails if its first two arguments can never be unified.

5.3.1.3. Clause Process

On executing an untrusted clause process, the emulator sees whether the clause’s goal can ever be head matched with the clause head. If not, the clause process is failed and any offspring killed off. A successful head match is recorded in the clause process’s status field. If the clause process’s goal needs to be further instantiated to satisfy the head argument matching requirements, the clause process will be suspended upon these variables. Suspension ring attachments are recorded in the same way as for primitive goal processes.

If the head arguments have been newly matched, child processes are spawned to evaluate the guard goals unless the guard is empty. If the guard is empty, the clause process commits immediately and spawns processes for each body goal. If no body goals exist, the clause process succeeds at once. If head arguments have been matched and guard or body goal processes have been spawned, the process suspends upon its children. If head arguments have not been matched, the untrusted clause process will be suspended upon relevant variables instead.

Where the guard is not empty, the clause process goes to sleep until its count of its children reaches zero. When the count becomes zero, the clause process is awoken from being suspended on its children and scheduled. On being executed, the clause process trusts. On trusting after terminating any clause rivals and their offspring, the clause process promotes itself to its parent. It then creates and schedules its body goal processes and goes to sleep again suspended upon its body goal processes. When the trusted clause process’s count of its children reaches one, the remaining child process is promoted to serve in place of its parent. When the child count reaches zero, the clause process signals success to its parent and then terminates itself. Signaling success to its parent involves decrementing by one its parent clause process’s count of its children. If the parent has trusted, and has only one child left, that child is also promoted in place of the parent.

Should a child of a clause process fail then this results in the clause parent being killed off along with all its spawned offspring. This killing also continues up the process tree until an untrusted clause process is reached which has rivals or could have rivals if they were scheduled. Should the attempt be made to kill off a parentless process, the whole evaluation of that process tree ceases immediately and failure is reported to
its meta-parent or at the top level.

In the interpreter prototype runtime variable correlates of clause code variables used in the clause’s head or in the clause’s guard, which also occur in the body of the clause, are stored by the clause process in a chain of data records when guard goal processes are created. This enables the clause process to create body goal processes with the right variables common to both head, guard and body. This information has to be stored because body goal processes are not created at the same time as guard goal processes in order to keep process creation demand driven. Compiled implementations avoid this overhead by compiling such variable information into the compiled code directly. Information on the location of child processes is available in the Relatives fields of the process descriptor. It enables the clause process to locate its offspring when killing them off. Information on search parallel clause rivals prior to trusting and on sibling conjuncts after trusting is also available in the Relatives fields. In this way or-parallel rivals can be killed when the clause process trusts. This information can also be used to kill off and-parallel siblings should the process fail after trusting.

5.3.2. The Data Storage Model

The data storage model has been devised to try to

- constrain peak memory usage
- recycle early reusable space

Both are important where infinite producer processes and long running programs are executing.

5.3.2.1. Terms

Clause code and run-time data structures are stored as tagged four byte words. The tag is four bits long. A special bit is used by the garbage collector and suspension rings. The other twenty seven bits stores the content. Variables, atoms, integers, real numbers and the nil list all occupy one word. Lists and structures occupy several contiguous words. System atoms are used to represent built in predicate atoms. They save on symbol table accesses and are used for fast built in predicate identification. Special extra cell types are used to represent clauses and non-existent values. The details of the data structures are summarised in Table 5.1. This scheme of representation uses at least eight different tags.
<table>
<thead>
<tr>
<th>Element</th>
<th>Tag</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>1</td>
<td>address of a cell or a frozen integer value</td>
</tr>
<tr>
<td>atom</td>
<td>2</td>
<td>address of string value</td>
</tr>
<tr>
<td>system</td>
<td>3</td>
<td>unsigned 27 bit integer atom value code</td>
</tr>
<tr>
<td>integer</td>
<td>4</td>
<td>signed 27 bit integer</td>
</tr>
<tr>
<td>real number</td>
<td>5</td>
<td>signed double length real number address</td>
</tr>
<tr>
<td>nil list</td>
<td>6</td>
<td>undefined</td>
</tr>
<tr>
<td>list</td>
<td>7</td>
<td>address of tail cell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell+1 contains head</td>
</tr>
<tr>
<td>structure</td>
<td>8</td>
<td>unsigned integer giving arity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell+1 contains functor name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell+2 contains first argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cell+n+1 contains nth argument</td>
</tr>
</tbody>
</table>

Table 5.1 Tagged Data Structures

The interpreter prototype handles variables using structure copying except with arithmetic primitives where structure sharing makes better sense. Only in such contexts is the content of a variable given the second interpretation [Taylor 1987]. More parsimonious logic programming implementations have reduced the number of tags to four [Gee et al 1987] - variables, constants, lists and structures. A conceptual advantage of the scheme above is that a different tag is used for each of the main kinds of object. A practical disadvantage is that its multiplicity of tags slow up unification and related term matching routines. Gee’s four tag scheme has the advantage of packing integers, atoms, and the nil list into one tagged constant category which can be treated by such routines as a single type of thing. Clauses are doubly linked together with separate pointers to

- head of clause
- guard
- body

They are accessed through a relation table which also records their source language. Four extra fields record

- head argument variables
- number of head argument variables
- body variables which also occur in the head or guard
- number of body variables which also occur in the head or guard
In a compiled implementation it would be unnecessary to store such information. Parlog relations also have an extra pointer pointing at the mode for a clause which is stored as a structure in the code area. This has no significance for emulation only for translating Parlog to the lingua franca.

5.3.2.2. Records

Two data elements of constant size are used for recording the run time execution state

- process descriptor
- data record

A process descriptor with a fixed number of fields, currently twelve, records the nature and characteristics of spawned processes. A tree structure of processes is generated as execution proceeds. This process tree is stored as a heap and grows. A subset of this tree is linked into the runnable process ring. Other parts of the tree are doubly linked rings of processes attached to a variable upon which they are suspended. Process descriptors are allocated and released as they are required. Freed descriptors are placed upon a linked freed storage space list. New processes are allocated storage space from any unused process descriptors on this list before new space is acquired from the process heap.

The other main run time data structure is a data record with an odd number of fields each one cell long. Whatever length it is given, determines the maximum arity of a stored or calling argument literal which can be handled. Its length can be made to depend on the maximum arity of currently stored clauses at the start of a top level CLP engine invocation or it can be set to a fixed value in the fashion of SPM’s 16 cell limit to its similar a-vector data structure [Gregory et al 1989]. A data record is used to record

- goal’s arguments
- shared body variables
- runtime values for clause code variables
- suspension ring attachments

The last cell of data records used to record suspension ring attachments is used by the data record to chain data records together. A heap of data records is generated as execution proceeds. The same heap is used by the process tree as is used by the data record heap. The process tree grows upwards and the data record heap grows downwards. Freed data records space are reclaimed by the use of a linked list of data records
which are allocated space but not assigned to a process. The following items

- integers
- floating point numbers
- nil list
- atoms
- variables (local)

have single valued entries in a data record assigned to a goal or a trusted clause process to record the values of calling arguments. Other terms like

- variables (global)
- lists
- structures

are stored on the heap. Variables, which are shared between goals in a clause and which do not occur in the head, are recorded in unused cells in the arguments data record for that clause or point at a cell on the global heap, if no unused cells in the arguments record are available. Lists and structures are built on the global heap under the rule that variables on the heap do not point out of the heap. Garbage collection needs to be performed on the global heap, because it will grow as execution proceeds and not all space on it will be reclaimed. Data records are not shared between processes but may be passed from process to process.

A suspension ring pointer points at two cells on a chain of data records hanging off a process. These cells are doubly linked to two cells on other data records hanging off other processes suspended on that variable. These data records have a back pointer to their owning process. Figure 5.4 shows two processes A and B suspended on variable X and three processes A, C and D suspended on the variable Y. The backpointer from the data records to the owning process is not shown.

In this way multiple processes can be suspended upon multiple suspension rings. Whenever a variable is bound or shared, its corresponding suspension ring, if any, is scheduled. Possession of a suspension ring is indicated by the garbage collection bit being set. The suspension ring for a variable is located by using the suspension hash table. Variables with their suspension bit set, have an entry in the suspension table made by hashing on the address of the variable cell. The suspension table entry records details on the ring of processes suspended on that variable. A more memory compact solution to suspending processes on variables
might have been adopted by using a special tagged data object to represent a variable with an attached sus-
pension ring. The content of that memory cell could then have pointed at the first process in the suspension
ring. This approach has been used in SICStus Prolog [Carlsson 1987] as well as by SPM [Gregory et al
1989] p.400-401. The disadvantage would be the introduction of a new tag which would have to be spe-
cially handled in all the unification routines.

5.4. Scheduling

The fairness of execution of concurrent logic programming languages can be characterised from the point
of view of clause selection fairness and from the point of view of concurrent resolution fairness. The emu-
lator’s scheduling is
• depth first
• bounded
• lazy

The scheduling is depth first because as goal processes spawn new clause processes, and as clause processes spawn goal children, these new processes are normally added to the next runnable places on the runnable process ring. It is bounded, because that is needed for balanced scheduling where infinite producer processes are possible. The scheduling is lazy, because only one clause process at a time is created and scheduled. Another clause process will be created and scheduled if a prior clause process fails. It will also be created if its busy waiting goal parent is run again, because none of the current clause process children have trusted. This scheduling scheme is biased towards the earlier clauses for a relation. This bias enables users to improve the performance of the execution of their programs by ordering the clauses for a relation which are to be searched in parallel. Clauses which will be selected more often should be placed higher in the listing of clauses for that relation.

Bounded depth first scheduling is used for the same efficiency reasons that it is widely used by other concurrent logic programming language implementations [Shapiro 1986]. Under this scheme processes are allocated a quota of resources which represent execution steps allowed before a context switch. The resources of a process are decremented each time it is executed. Spawned processes inherit the scheduling count of their parents less one. When a process’s quota is used up, it is allocated a new complete quota, probably several times ten reductions worth. It is then passed to the last position in the runnable process ring. All user-defined processes belong to a ring of runnable processes, are attached to a suspension ring correlated to a variable or are suspended upon a child. When guard processes are created by an execution step, they are scheduled immediately unless context switching occurs. No checking is done to see whether they should be attached to a suspension ring. This policy is in line with the findings of [Kishimoto et al 1987]. A similar policy applies to the scheduling of body processes when their parent trusts. System and untrusted clause processes may be attached to more than one suspension ring. When a variable on a suspension ring is instantiated or shared with another variable, all processes upon its suspension ring are scheduled for immediate execution ahead of any other processes. This is again in accordance with the findings of [Kishimoto et al 1987]. Furthermore a process suspended upon an input stream or upon a variable, which is woken up, is given a new quota of resources. This also biases execution towards newly woken primitive
goal processes.

Having an associated suspension ring is indicated by whether the status bit of a variable cell is set or not. Suspension rings are pointed at from a suspension table. The address of the root variable is used to hash/index into the suspension table. Each entry in the suspension table has three fields which are either unset or point at the variable and the ends of the suspended process ring respectively. Hashing collisions are disambiguated by means of the backpointer to the relevant variable on which the processes are suspended.

Parents suspended on children wait until they are woken by their last child succeeding or failing. Goal processes wait busily and schedule one new clause offspring per run until no more may be scheduled. Then the goal process suspends upon its children and only awakens if a child clause process fails or trusts. The scheduling system takes no account of how much time executing a primitive goal process takes so long as the primitive goal process does not spawn any offspring in being satisfied or failing. This is less even handed when compared with scheduling using time slices. However, it is simple to manage.

5.5. Input and Output

The emulator supports asynchronous input for the primitives

- get0(Ascii, File)
- skip(Ascii, File)
- read(Term, List, File)

The three basic asynchronous input primitives input characters, skip input until just before a specified character or type of character is available, and read terms. Full details are given in appendix B. The emulator also supports asynchronous input conducted under the auspices of the meta-primitives instream/1 and call/3. When a call to an asynchronous input primitive is made, that call is processed immediately, if input to satisfy it is available. If no input to satisfy it is available, it is hung in order on a queue of unprocessed input primitive goal processes associated with each open file. Each of these processes keeps a record of its meta-parent. This is to enable identification of the appropriate active process ring upon which to schedule the process.

Asynchronous input is supported underneath by setting an event flag when the Unix signal SIGIO is received. At the start of the next emulator cycle this event flag is checked. If it is set, all open files with
suspended processes waiting upon input are checked. If there is input available on a file upon which processes are suspended, the calls are progressively processed. This carries on until either there are no more input processes left to be processed or until the lead process requires more input than is available. The next open file is then examined and handled until all open files have been dealt with. This processing only results in input primitives being satisfied with appropriate bindings being made or their failing to be so. Either way input primitive goal processes are scheduled. Only when they are run will such processes succeed or fail with consequences for their relatives on the process tree. A special ring is formed of control meta-calls which are suspended upon their offspring where one or more offspring are suspended waiting upon input in that tree. Each time the SIGIO flag is set and asynchronous input is handled at the start of an emulator cycle, these meta-calls’ control blocks are scanned to make sure that none of their offspring have woken up. Should this happen, the meta-call goal process is scheduled on its meta-parent’s runnable process ring. A significant advantage of having a high level primitive read/3 which can parse input as Prolog terms as an atomic operation is its performance. Its asynchronous behaviour also makes it easy to handle input from multiple windows just by reading from the window devices directly.

5.6. Conclusion

The lingua franca of chapter four is executed by and-parallel input resolution. This chapter has sketched an execution model for it and a scheme for implementing that model. The approach shares many features with implementations of Parlog like SPM [Gregory et al 1989] and the JAM [Crammond 1988], except that the implementation model is not articulated like SPM and the JAM as an abstract machine with its own instruction set to which Parlog is compiled, although it could be. It has been presented as a uni-processor execution model like SPM without description of features for multi-processor execution like the JAM. Like the JAM the execution model features two kinds of process, goal and clause, rather than the four process types, sequential AND, parallel AND, sequential OR and parallel OR processes of SPM. Unlike the JAM and SPM there is no implementation support for sequential conjunctions (the lingua franca does not support the construct). Unlike SPM but like the JAM the execution model presupposes head argument matching strictly precedes guard evaluation. Like the original SPM but unlike the JAM input matching is performed concurrently and not sequentially.
Like SPM and the JAM complex argument terms are stored in the heap. Like SPM simple goal arguments are stored using fixed sized memory blocks called data records (a-vectors in the SPM) from a special heap, rather than allocating cells of memory on an argument stack like the JAM. Like SPM space recovery of these data records is done when the owning process succeeds or fails by adding them to a free list. However, unlike SPM and the JAM which allocate Heap space for them, logical variables in the body of a clause, which do not appear in the clause head, are stored with their parent process’s calling arguments to save on Heap usage. A delayed process promotion scheme ensures this does not leave dangling pointers. It is sustained in the overall scheme by eschewing variable pointer chain short-circuiting upon successful head argument matching and by delaying full promotion of processes representing body goal processes on top of their grandparent goal process. It trades off a dereferencing time optimisation against a significant reduction in memory cells which need to be garbage collected. The JAM adopts the different scheme of allocating goal arguments on an argument stack from the top rather than using fixed length blocks of memory taken from a special heap. As JAM argument cell deallocation can happen anywhere on the stack, holes of deallocated memory cells must await surfacing to the top of the stack or compaction during garbage collection before they can be reallocated.

Like SPM but unlike the JAM parallel clause process (parallel OR) creation is lazy. Unlike SPM, but like the revised version of the JAM, processes can suspend on multiple variables at once. Like SPM but unlike the JAM parent processes store pointers to their offspring. This allows parents to locate all their offspring, and allows meta-calls to use these pointers to suspend all their offspring. Like SPM but unlike the JAM a bound (which can be dynamically changed) is placed on depth first scheduling. Other differences from the JAM or SPM include direct support for high level IO primitives, primitives for atomic unification and forms of unidirectional unification, and a resource count rather than a time slice scheduling system. The user’s view of a working prototype, which implements the CLP lingua franca and the atomic interface from the CLP engine to Prolog, is described in appendix B.
Chapter 6

6. Escape Exception Handling in Prolog

6.1. Introduction

Chapter three proposes an architecture, which couples Prolog and CLP resolution engines, to solve the problem of how to program in logic for parallel execution both the systems and the deduction aspects of knowledge based systems. Chapter four describes and justifies a suitable CLP language to execute on the CLP resolution engine, and chapter five deals with implementation considerations for the CLP resolution engine. This chapter deals with extensions to the Prolog language required for supporting chapter three’s coupling interfaces and with the question of how to implement them on the Prolog resolution engine. The architecture of chapter three envisages using a conventional WAM engine to execute Prolog. It has to incorporate two main extensions, one for handling persistent delayed goals created by \texttt{async/2}, and the other for handling escape exceptions. Persistent delayed goal creation by \texttt{async/2} is merely a variant of delayed goal creation by the \texttt{freeze/2} primitive, whose implementation has already been adequately dealt with by [Carlsson 1987] for WAM based Prolog engines. This leaves escape exception handling in Prolog to be tackled. Two issues need to be addressed - the semantic question relating to what escape exception handling offers Prolog, and the implementation question of how to support escape exception handling efficiently in a WAM engine.

6.2. Exceptions and Choice Points

Prolog would benefit from having scruetable control constructs with declarative meaning for dealing with exceptional cases of evaluation and for tolerating execution limitations of software systems. Exception handling facilities aspire to provide this. Existing Prolog systems already allow calls to primitives with anomalous arguments to be overlaid with a call to a programmable exception handler. However, this \textit{notify} style of exception handling

\begin{itemize}
  \item localises anomaly management in the evaluation tree
  \item restricts its application to primitives
  \item does not make Prolog systems more robust to Prolog engine faults
\end{itemize}
A more serviceable kind of exception handling can be created by reworking the conception of the *remote cut* and exploiting the choice point mechanism already used for backtracking in canonical implementations of Prolog. Programs can be allowed to create named exceptional choice points and an exceptional form of backtracking can be allowed to fail computations directly back to these named exceptional choice points. In this way anomalies become exceptional cases which inference handles as exceptional alternatives. Application programs can be made more reliable by arranging for the Prolog system to backtrack in this fashion on encountering a particular execution error, enabling the application program to set handlers to catch that type of error.

The scope for supporting exception handling in Prolog is revealed by unexploited possibilities in the functionality offered by choice points. A choice point is a navigational sign post in a Prolog computation which records its state. As D.H.D. Warren observes

"A choice point contains all the information necessary to restore an earlier state of the computation in the event of backtracking." [Warren 1983] p.3.

However, canonical Prolog implementations make rather limited use of this power of restoration. They are only concerned to reset the computation to its former state, in order to resolve the current goal instead with the next clause, whose head unifies with the goal. This limited use of powerful functionality, which can restore earlier states of a computation, invites being generalised and developed in a way which would enable more flexible use of choice points. A general escape exception handling facility and new control structures could then be built out of existing choice point functionality with only slight modification to canonical Prolog systems. The extra functionality would conservatively extend existing functionality. Normal Prolog execution would happen by default, and only where the extra choice point facilities are invoked, would the flow of execution be extended in extra ways.

In canonical implementations of Prolog choice points are only

- *created* during execution where another satisfier of the goal exists
- *revisited* upon the failure of a goal
- *backtracked* to in the reverse order of creation
- *enabled* to offer the alternative of resolution with the next satisfier
However, each of these conditions of use could be relaxed. In addition to the normal creation of choice points during execution, it could be made possible to generate extra choice points which may be

- *created* explicitly by execution of a primitive
- *revisited* upon a user/system initiated event
- *backtracked* to directly despite being overlaid
- *enabled* to offer a user defined goal as alternative

A simple example of the use of exceptional choice points would be the incorporation of an execution abortion mechanism into an application program running on top of a Prolog system. The application program could be some kind of interactive system based on a simple forward recursing call which issues a prompt, 

reads a command, executes it producing a response, reports the response and then begins the cycle again.

(1) interact :- write(’ ’),
read_command(Command),
execute(Command, Response),
report(Response),
interact.

A command abortion capability for such an interactive system can be programmed on a Prolog system provided that it

- initiates backtracking when it receives a break signal from the user’s terminal
- backtracks to the last exceptional choice point named *break*

This could be achieved by amending the clause to

(2) interact :- write(’ ’),
otherwise(break, interact),
read_command(Command),
execute(Command, Response),
report(Response),
interact.

A new metacall primitive *otherwise/2* has been introduced, which is not ordinarily resatisfiable. On being satisfied it creates an exceptional choice point named by its first argument and with its second argument as alternative upon being exceptionally failed back to. This choice point is additional to ordinary choice
points but is ignored during normal backtracking. However, should the user issue a break signal at the terminal while the Prolog application is executing a command (e.g. during the attempt to satisfy the `execute/2` goal), then the Prolog system will backtrack through prior choice points pausing only to consider the names of exceptional choice points until it encounters an exceptional choice point named `break`. This becomes the point at which forward execution recommences with the exceptional choice point’s alternative goal. Thus during normal execution either forward or with backtracking (2) behaves like clause (1). During exceptional backtracking (2) behaves like clause (3). Exceptional backtracking, initiated by a break signal during the execution of the middle four literals in the body, results in the `otherwise/2` alternative, the last literal, being selected, so long as no other exceptional choice point named `break` is encountered beforehand.

\begin{verbatim}
(3) interact :- write('> '),
    ( read_command(Command),
      execute(Command, Response),
      report(Response),
      interact ;
      interact).
\end{verbatim}

In this way a simple command abortion mechanism can be integrated with an interactive command execution loop†. This kind of behaviour cannot be achieved by responding to break signals by failing the current goal. Doing this would not ensure that backtracking initiated by a break signal had to continue to any specific earlier choice point in the computation. Nor would it specify that any particular goal had to be satisfied upon ceasing to backtrack under such a stimulus. A construct like this, which is somewhat similar to Lisp’s `catch` and `throw` functions, is needed to ensure that the break exception propagates to and only to an earlier handler of the appropriate type.

Because of the forward recursing nature of this `interact/0` program, new exceptional choice points called `break` would be built on the top of the stack each time round the recursion loop. They do not overwrite earlier ones of the same name. To prevent this accumulation, a cut could be inserted before the last goal of `interact/0`. The effect of this would be to prune back the stack to the last ordinary choice point encountered before reducing the goal `interact` to the body of its clause. The exceptional choice point created by executing the `otherwise/2` goal would get pruned along with any other choice point created later than this ordinary choice point (e.g. those created by satisfying `read_command/1`, `execute/2` etc.).

† Another approach to handling terminal interrupts would be to use extensions to Prolog for handling `freeze/2` [Carlsson 1987].
Exceptional choice points can be used for handling

- escape exceptions
- recovery
- execution errors

When evaluation of a goal is abandoned, because unusual circumstances render it inappropriate to continue, it is sometimes convenient to raise an exception which signals this unusual circumstance. The exception can then be handled at some particular point further back, rather than simply by failing the problematic goal. Exceptional choice points can also be used for recovery from unwanted outcomes or wayward events by restoring earlier states of a system. This can involve rolling back the state of the database. Thirdly, escape exception handling is useful for coping with execution errors.

6.3. Exception Handling

In analysing exception handling facilities [Goodenough 1975] Goodenough distinguishes between

- notify exceptions
- escape exceptions

Notify exceptions suspend performing the operation raising the exception, handle the exception and resume the operation. Escape exceptions on the other hand terminate the operation raising the exception and continue execution in a separate context with the handler for that exception type. Each style of exception handling has merits, as do combinations of both. Notify exception handling has been added to Prolog with considerable effect in KL0 [Chikayama 1983] and MU-Prolog [Naish 1986] p.95. The exceptional goal is overlaid by a call to the error handler which includes full details on the anomalous call. This allows user programs to intercept and handle the error or to leave it to the default handler. Escape exception handling has not been added to Prolog systems, principally because Prolog can already be used unamended for escape exception handling. The argument of this chapter is that Prolog programming could be usefully improved by being able to express escape exception handling explicitly and by having more apposite tools for doing so.
6.3.1. Escape Exceptions

Escape exception handling is used to considerable effect in the functional language ML [Harper et al 1986], and in the imperative language Ada [Pyle 1981]. ML and Ada allow evaluation to be abandoned, when it cannot be carried through in the intended manner, and signal this fact by raising a named exception. This allows a tricky boundary case of evaluation to terminate the execution of the current operation. The flow of execution is returned to a higher level from which it was invoked, or is passed to a different context at the current level, where it may be dealt with more easily. In control applications escape exception handling allows an anomalous state of a monitored device to be responded to in an exceptional manner by signaling the anomaly and switching the flow of control to where it may be more readily handled.

An example of the use of a built exception break has already been given. It illustrates how simply choice points and Prolog’s backtracking mechanism can be used to escape from the current context of execution back to a point at which the exceptional circumstance can be handled. In order to give fuller range to raising escape exceptions in Prolog, a new primitive fail/1 is introduced. On encountering the call fail(<atom>), the Prolog emulator is required to fail this goal and to begin exceptional backtracking just as it would do with the built exception break. This carries on through prior choice points unstacking everything in between until the emulator encounters the first exceptional choice point with the name <atom>. The alternative at this point becomes the new goal.

Languages like ML and Ada, which embody escape exception handling, allow three things to be done with exceptions. Exceptions may be declared, raised and handled. Exceptional choice points enable Prolog to handle escape exceptions in an analogous way. Exceptions can either be

- built in
- user defined

User defined escape exceptions are

- declared by being named in fail/1 and otherwise/2 goals
- raised by executing a fail/1 primitive
- handled during exceptional backtracking at exceptional choice points of the exception’s name

and built in exceptions are
\begin{itemize}
\item \textit{declared} by being specially known to the system in advance
\item \textit{raised} in their exception context by the system or explicitly by a \texttt{fail/1} primitive
\item \textit{handled} in the same way as for user defined exceptions
\end{itemize}

Unlike exceptions in ML and Ada, these Prolog exceptions do not pass back exception values. Their use lies in their failure for a reason with a name, not in successfully returning exception values. The only thing, which an exception can pass back, is its name. This will happen if the exception name is not instantiated when the exception handler is set up. In this case the handler \texttt{otherwise(Name,Goal)} will catch the first propagating exception and will bind \texttt{Name} to that exception’s name. Exception failures \textit{propagate} until they find their appropriate handler by virtue of the exceptional backtracking mechanism which continues backtracking until an \texttt{otherwise/2} goal for that exception type is found. Failure to find a handler results in the whole computation failing. A more expressive variant on this scheme [Michaelson & Gregory 1990] would allow escape exceptions to pass back atomic values by adding a second argument to \texttt{fail}. Exception handlers set up by an \texttt{otherwise} predicate with a third argument would have that third argument unified with that value if the exception handler catches the propagating exception.

\section*{6.3.2. Prolog and Escape Exceptions}

The case for this extra functionality can be made by examining how standard Prolog copes with escape exception handling. A simple example of escape exception handling is given by arithmetic with unknown numeric values. One way of evaluating an arithmetic expression with an unknown value in it is to evaluate the whole arithmetic expression containing it as an unknown value. This can be done by raising an escape exception on encountering an unknown value in an arithmetic expression, and handling the exception by making the unknown the value of that expression. The atom \texttt{null} can be used to represent a particular but unknown numeric value. The problem can be cast as that of defining a clause for an infix relation \texttt{:=/2} which functions like the primitive \texttt{is/2} but handles unknown values, \texttt{nulls}, occurring in place of numbers as well. This can be done using an auxiliary infix relation \texttt{sums/2} in standard Prolog as follows.
A := B :- A sums B.
null := _.

A sums A :- number(A).
A sums B :- B =.. [Op, C], D sums C, E =.. [Op, D], A is E.
A sums B :- B =.. [Op, C, D], E sums C, F sums D, G =.. [Op, E, F], A is G.

This predicate is called instead of is/2. When the predicate sums/2 is called with its right hand argument instantiated to null, no clause can be found to satisfy it. The exceptional case results in the current goal failing. This failure propagates all the way back to the alternative provided by the second clause for :=/2. It provides a fallback satisfier which unifies the first argument with null.

This way of programming escape exceptions is serviceable and popular. However, it has several drawbacks.

- exception raising is implicit
- handler resatisfies irrelevant search
- deep backtracking is hard to control
- fails to discriminate among different exceptions

The first drawback is that the exception raising is implicit. No clause actually raises the escape exception. This cannot be adequately remedied by adding a redundant clause

A sums null :- fail.

to the definition for sums/2, and staying the parsimonious programmer’s knife. The new clause still would not make clear that an escape exception was being raised rather than an alternative eschewed. The same applies to the handler clause. Neither unambiguously states its function. Thus the clauses do not wear on their face the intention in their functionality to raise and handle escape exceptions. This does not make for scrutable programming.

A second drawback is that this method makes the predicate in question resatisfiable for unexceptional cases. In the example above failing back into :=/2 succeeds again with the fallback solution. This does not undermine the practicability of the method. Judicious use of cuts can alleviate the difficulty. A cut can be inserted at the end of the first rule for :=/2. However, having recourse to using cuts in this way is not conducive to securing clarity of meaning.
A third drawback is that sometimes exception propagation has to use deep backtracking. In the example above this is not problematic, but in general it is. Ensuring that with deep backtracking the emulator arrives where it is wanted and does not succeed somewhere before then, is not always straight-forward, especially where recursive predicates are involved. Literals may need to be rearranged, extra clauses interpolated and cuts inserted to ensure that deep backtracking arrives only where it is wanted. This extra complexity does not contribute to semantic perspicuity. Using backtracking in a dual role both to explore alternatives and to propagate exceptions, also makes constructs, which control backtracking, ambiguous as to their function. The proper understanding of how a program with exception handling works, requires that the reader knows which function each control construct like a cut is performing in the program. Since this function is not worn on its face, understanding what is happening in the program is made harder.

Fourthly where several different exceptional cases have to be handled at once, the method illustrated above lacks the means to distinguish among them. In an arithmetic application unknown values, division by zero, arithmetic overflow, uninstantiated values and arithmetic underflow may all have to be handled at the same time. Disentangling the handling of different types of exception from each other, when all exceptions can only be uniformly raised by goal failure, is hard.

All four drawbacks contribute to a loss of scrutability of meaning and to an increase in clause complexity. A simpler and more manifest way of avoiding all these drawbacks and achieving the same effect can be had by adding a new clause to \textit{sums}/2, to raise the exception explicitly using the newly introduced primitive \textit{fail}/1:

\begin{verbatim}
A sums null :- fail(unknown).
\end{verbatim}

The two clauses for \texttt{=:}/2 can be replaced by the single clause

\begin{verbatim}
A := B :- otherwise(unknown, A = null),
A sums B.
\end{verbatim}

having the declarative reading

\[
\forall A \forall B \exists E ( \text{exception}(E) \& E = \text{unknown} \& A = \text{null} ) \vee \text{sums}(A, B) \supset :=(A, B)
\]

where \textit{exception}/1 is true of a thing when and only when that thing is an exception. Executing the call \textit{fail(unknown)} raises the exception \textit{unknown}, which is handled using the \textit{otherwise}/2 predicate for \textit{unknown}.
The anomalous case is required to cause exceptional backtracking, which carries on until an appropriate alternative satisfier for handling the exceptional case is encountered. This program wears on its face its recourse to escape exception handling. The exception handling clause has a declarative reading, and is not resatisfiable for non-exceptional cases. Exception handling is easily extensible for handling multiple types of exception. It does not impose any burden on the programmer to ensure that an exception propagates correctly, and it avoids loading clauses with extra cuts.

If there was a built in escape exception illegal_argument, which was raised when finding a non-numeric operand in an arithmetic expression that was being evaluated, then it would be possible to use a single clause to achieve the same effect.

\[
A := B :\text{ otherwise(illegal_argument, A = null),}
\]

\[A \text{ is } B.\]

This clause is less discriminating than the original clause. All non-numeric operands encountered in evaluating an arithmetic expression are treated uniformly. It has the extra inconvenience of catching all propagating illegal_argument exceptions. These might arise from illegal uses of irrelevant primitives.

6.4. Recovery

Exceptional choice points are rather serviceable for handling recovery from an undesired outcome or an untoward happening. Sophisticated command driven interactive systems need to be able to restore the state of the system as it was one or even several steps before. This requires an undo support facility [Yang 1988]. The user may regret having brought about the current state of things. The wrong command might have been issued or the wrong keys pressed. A user might even have changed his mind about accepting what he has brought about, or have decided in the light of an outcome that he wants things to go back to the way they were before. Thus an interactive command driven system which a user feels is safe to use, that encourages learning through exploration, and try and see strategies of use needs to be able to restore the system to a state it was in previously.
6.4.1. Recovery without Side Effects

Interactive trace facilities in Prolog are one example where such a recovery facility would be of value. Thus in a Prolog trace a user may use a `skip` command to skip interactive tracing until a goal is executed to completion. The user may then decide in the light of the outcome that he wants to restore the Prolog computation to the state it was in before attempting to satisfy that goal to completion. Even discounting the use of Prolog predicates with side effects, Prolog’s normal backtracking mechanism cannot be used to restore such a prior state in one step. Goals are often resatisfiable and so initiating backtracking, in the hope of backtracking all the way to the point at which the `skip` instruction was issued, is not guaranteed to get there in one step. A subgoal encountered en route might be resatisfiable, and the flow of execution might be redirected forward again during the attempt to go all that way back (C.F. [Carlsson & Widen 1988] p. 21). On the other hand the mechanism of exceptional choice points is guaranteed to fail back the computation to a named earlier exceptional choice point in one step and that makes such recovery of earlier states, which have been transformed without side effects, simple to program using this functionality. A simple command driven interactive system programmed by the clause

```
interact(State) :- read_command(Command),
exe cute(Command, Response, State, Newstate),
report(Response),
interact(Newstate).
```

executes commands over a system defined by `State`. It updates `State` each time to `Newstate` without side effects. It might be given an undo capability by being replaced and augmented by an extra clause for `execute/4`.

```
interact(State) :- otherwise(prior_step, interact(State)),
exe cute(Command, Response, State, Newstate),
report(Response),
interact(Newstate).
```

```
exe cute(undo,_,_,_) :- fail(prior_step).
```

On executing the command `undo`, the exception `prior_step` is raised and causes failure back to the last exceptional choice point named `prior_step`. Intermediate choice points, which represent alternative
satisfiers for subgoals of `execute/4`, are ignored. The alternative goal `interact(State)` at this exceptional choice point becomes the new goal which the emulator attempts to satisfy. This capability of undoing is multi-step, in that it can be continued back to the initiating call which invoked the interactive system.

**Database Rollback**

Prolog’s standard database manipulation primitives `assert/1` and `retract/1` are not fully reversible. However, they can easily be extended by an extra argument to enable all changes to the database to be recorded in a history list, which is marked at the beginning of each interactive cycle. Changes to the database for the last cycle’s entries can then be reversed using this history list upon handling each undo exception. The clause manipulation primitives `retract/2` and `assert/2` unify their second argument with the numerical position of the clause in the relation, and if it is uninstantiated, `assert/2` asserts to the end of the relation and `retract/2` retracts as close to the front of a relation as possible. They can be used to support the derived primitives `retract/1`, `asserta/1` and `assertz/1` normally used by users to manipulate the database as follows:

```
retract(Clause) :- retract(Clause, N), update_history(retract(Clause, N)).
asserta(Clause) :- assert(Clause, 1), update_history(assert(Clause, 1)).
assertz(Clause) :- assert(Clause, N), update_history(assert(Clause, N)).
update_history(Item) :- retract(history(H), _), assert(history([Item|H]), 1), !.
```

By checkpointing the history list at the start of each cycle by adding a marker, it becomes possible to restore the changes to the database made in each cycle.
interact(State) :- update_history(checkpoint),
read_command(Command),
perform(Command, Response, State, Newstate),
report(Response),
interact(Newstate).

perform(undo,_,_,_) :- fail(prior_step).
perform(C, R, S, N) :- C \= undo
otherwise(prior_step, (restore_database, interact(S))),
execute(Command, Response, State, Newstate).

restore_database :- retract(history(H), _),
restore_database(H, I),
assert(history(I), 1), !.

restore_database([], []).
restore_database([checkpoint|T], T).
restore_database([assert(Cl, N)|T], L) :- retract(Cl, N),
restore_database(T, L).
restore_database([retract(Cl, N)|T], L) :- assert(Cl, N),
restore_database(T, L).

So long as the program confines itself to using asserta/1, assertz/1 and retract/1 to manipulate the database, the trailed history of changes to the database can be used by the goal restore_database to restore the database as it was back at the start of each interactive cycle.

6.4.2. Transaction Control

This restoration capability can be adapted to support a general transaction handling facility for supporting concurrent access by two lightweight Prolog processes (Sun OS 4 Unix) to a shared clause database on a single processor machine. The definitions of restore_database/0 and update_history/1 need to be revised to distinguish between the histories of different execution threads. This can be done, using a primitive process_id(Id) which unifies its argument with a unique identifier of the (lightweight) process executing it, as follows:
restore_database :- block_context_switch,
                      process_id(Id),
                      retract(history(H, Id), _),
                      restore_database(H, I),
                      assert(history(I, Id), 1), !,
                      enable_context_switch.

update_history(Item) :- process_id(Id),
                        retract(history(H, Id), _),
                        assert(history([Item|H], Id), 1), !.

Each history list is uniquely identified with the execution thread, whose changes to the database it records. Two primitives `block_context_switch/0` and `enable_context_switch/0` ensure context switches do not themselves interfere with the restoration process. Another primitive `thread_slice/1` is also introduced. It unifies its argument with a unique identifier of the current slice of the executing (lightweight) process. The current slice exists only between successive switches by the operating system to executing different execution threads (lightweight processes). Critical interactions with the database can be run within a transaction metacall predicate `transaction/1` defined as follows†

```
transaction(Goals) :- otherwise(transaction_failure, (restore_database, fail)),
                    begin_transaction(No),
                    call(Goals),
                    end_transaction(No), !.

begin_transaction(No) :- process_id(Id),
                       thread_slice(No),
                       assert(history([], Id), 1).

begin_transaction(No) :- fail(transaction_failure).

end_transaction(No) :- thread_slice(No), !.
end_transaction(No) :- fail(transaction_failure).
```

`transaction/1` tests whether there has been a context switch to another concurrent execution thread to see whether to abort the current transaction. A context switch is detected by seeing whether the identifier of the current slice of the executing (lightweight) process has changed from what it was at the beginning. In that event an escape exception `transaction_failure` is raised, the exception is handled by restoring the database, and the transaction meta-call is failed. It is assumed that context switching is disabled from the moment an escape exception is raised until it is handled and at least two reductions have taken place. This restriction is

---

† See [Naish et al 1987, Massey 1988] for other approaches to transaction handling in Prolog.
necessary to give the escape exception handling mechanism a chance to execute a `block_context_switch` call if need be. Its effect will not last long and should not impose much overhead. If `call(Goals)` fails, the same exception is also raised with the same result.

### 6.5. Error Handling

Adding programmable recovery facilities to Prolog systems should enable the development of more fault tolerant and more reliable application programs. Various error recovery strategies have been implemented in Prolog systems like MU-Prolog [Naish 1986] and SICStus Prolog [Carlsson & Widen 1988], and they provide quite serviceable recovery facilities. However, they could be usefully augmented by an escape exception handling facility. If the Prolog engine was able to raise an appropriate built in escape exception on encountering an error, a handler embedded at an appropriate earlier point in the computation could catch the error exception as it propagated backwards, and provide an appropriate alternative goal to enable recovery from the error.

Built in types of escape exceptions, which it might be useful to have, include the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Exception Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>any built in exception</td>
</tr>
<tr>
<td>break</td>
<td>terminal interrupt signal</td>
</tr>
<tr>
<td>division_by_zero</td>
<td>division by zero exception</td>
</tr>
<tr>
<td>heap_overflow</td>
<td>global stack overflow</td>
</tr>
<tr>
<td>illegal_argument</td>
<td>illegal call to primitive</td>
</tr>
<tr>
<td>stack_overflow</td>
<td>local stack overflow</td>
</tr>
<tr>
<td>trail_overflow</td>
<td>trail stack overflow</td>
</tr>
<tr>
<td>unix_signal</td>
<td>serious Unix inter-process signal</td>
</tr>
</tbody>
</table>

Table 6.1 Built in Escape Exceptions

A Prolog system should offer users the option of having such conditions initiate exceptional backtracking to exceptional choice points with these names. This could be specified on setting up a Prolog system or in starting it up. Sometimes it may not be convenient to have hard wired exceptional backtracking upon these exceptional events happening. Simple success or failure of the goal might be more useful or a notify exception handler might be more apposite. Thus a Prolog user should be given the option (with a sensible default)
to switch on or off the raising of each built in type as an escape exception. An all-inclusive generic type of built in escape exception called any occurs in the above list. Runaway built in escape exceptions which have not been given a handler can be caught by it, or can be caught by a handler set up with a variable in place of its name.

6.5.1. Terminating Recursion

Just how usefully some tricky but common error conditions can be dealt with by escape exception handling can be shown by example. A common but important source of errors in Prolog is getting trapped in non-terminating recursive loops, and being unable to continue execution. Prolog employs SLD resolution in a left most depth first search for refutation of a query goal. Such search is not guaranteed to terminate. Notoriously such search can and does disappear depth first down infinite paths in the SLD resolution tree. In practice this means that memory exhaustion rapidly prevents the local stack being extended any further and normal computation cannot be continued. Canonical implementations regard non-termination as the programmer’s problem. On such implementations local stack overflow results in the computation being aborted. The rationale seems to be that as normal computation cannot be continued, all computation has to be aborted. The run-time stacks are reinitialised, and control is returned to the top level or to some default goal. This means that all information embodied in the computation’s run-time stacks is rendered inaccessible, and that whatever has been computed so far, but not asserted into the database, is lost.

If on memory exhausted stack overflow the Prolog system were instead to engage in deep backtracking to some pre-determined choice point, then significant space could be cleared on the run-time stacks. Forward execution might be able to commence again at that choice point with a suitable alternative goal without wasting what has been computed before that. Where one recovery leap backwards does not suffice, the pattern of recovery could be repeated again rolling the computation back to earlier and earlier nominated points until forward computation can succeed again. An example illustrates how this might be useful.
The clauses above for \textit{any/3} define a multiple solutions predicate like the classic all-solutions predicate \textit{findall/3} [Clocksin & Mellish 1981] p.152. \textit{any/3} takes a solution element template \texttt{X}, a goal \texttt{G} and unifies its argument \texttt{L} with a list of all satisfiers of the form \texttt{X} for each satisfier of \texttt{G}. It initialises the difference list of solutions and then calls \textit{more/3}. \textit{more/3} generates successive solutions by backtracking using \textit{collect/1} to add new solutions to the stored difference list.

It is assumed that the Prolog implementation raises the built in escape exception \textit{stack\_overflow} when local stack overflow occurs. This becomes useful over the database clauses

\begin{verbatim}
sibling(john, mary).
sibling(mary, jill).
sibling(A, B) :- sibling(A, C), sibling(C, B).
\end{verbatim}

when the following call is executed

\begin{verbatim}
| ?- any(X, sibling(john, X), L).
\end{verbatim}

\begin{verbatim}
L = [mary , jill | _8 ]
\end{verbatim}

\begin{verbatim}
yes
| ?- 
\end{verbatim}

The new all-solutions predicate \textit{any/3} collects all solutions like \textit{findall/3}. However, where local stack overflow occurs, as it does on the attempt to satisfy the subgoal \textit{sibling(jill,_)}, the Prolog system fails back the execution state to the last exceptional choice point for that exception and executes its alternative. Thus the goal \textit{retract(store(L-_-))} gets executed instead. The effect of exceptional backtracking is to free much of the
space claimed on the local stack and to enable execution to continue using the list of solutions computed so far. The fact that the computation has not been able to complete is indicated by the uninstantiated end of the list of solutions.

This example illustrates the power of using escape exception handling for recovering from errors. Not merely has a serious error been recovered from, but good sense has been made out of what was computed before the error occurred and out of the error’s occurrence. Other recovery strategies like notify exception handlers would not have been able to let the computation escape out of this predicament.

6.5.2. Robust Systems

Some programs are sufficiently important that it is essential that they be made robust to events like running out of memory and certain inter-process signals like stop and quit. This applies especially to long running programs. In such an event it is wasteful just to abort execution and give up all hope of making use of what has been computed so far. With memory exhaustion the program may only require more fast memory to continue with than the limit which the operating system currently allows the program. This can be achieved if the program is restarted within a different process with bigger limits to its allocated memory, or if the process is restarted on a bigger machine. Similar things apply to unwanted inter-process signals. It may be possible to restart the computation in a different process environment. Prolog systems like C-Prolog have a facility for saving the current state of a computation in the file File by using the primitive save(File). The Prolog state can be restored by issuing the Unix command

    prolog <File>

and execution continues with the goal immediately after the save/1 goal. This kind of facility can be exploited in a simple fashion to recover from critical events like memory exhaustion and the receipt of certain Unix inter-process signals as follows:
The original program uses periodic calls to `checkpoint/1` with the current version of the `Goal` at the time of call to mark opportune recovery points for the program. If the program encounters a serious event like running out of trail when executing, the built in escape exception `trail_overflow` is raised. This causes backwards recovery to the exception handler at the last checkpoint. The program state is preserved in the file `prolog_state` using `save/1`, and the program is halted if the process identifier is the same. The exception handler intercepts all escape exceptions, unifying the handler label `Name` with the exception’s name, but only hangs on to three types. All other escape exceptions are re-raised and passed back. The saved state in the file `prolog_state` can now be loaded into a process allocated rather more fast memory than before, or can be run on a bigger machine, or can be given a different process environment, and the program can be restarted at the point where it was last saved. The program recommences execution just after the save primitive. However, this time it fails the `process_id(Id)` call because the process identifier has changed and continues with the goal `call(Goal)` instead. Of course, such a facility could be hard wired into a Prolog system, but it would not be able to choose the opportune points at which to save the program state, nor to cater for any other special requirements. Thus the approach would be much less flexible.

### 6.6. Implementing Exceptional Choice Points

The Warren Abstract Machine is the canonical implementation of Prolog. A good introduction to it is available in [Gabriel et al 1985]. Modifications to its model serve to illustrate how the functionality for exceptional choice points can be implemented for Prolog systems in general. These will be briefly explained with reference to Warren’s original presentation [Warren 1983]. An enhanced version of Prolog-X has already implemented a restricted version of escape exception handling by different means [Massey 1989].
Exceptional choice points are created like ordinary choice points with a pointer referring backwards to the last ordinary choice point. An extra field, stored last in exceptional choice points, chains them by backward references to prior exceptional choice points. The emulator holds in a new register called the *exceptional backtrack* register, the address of the top exceptional choice point. This value is maintained up to date by the emulator.

Two new instructions are added to the WAM instruction set

\[
\text{rely}_\text{on}_\text{me}_\text{otherwise} \ L, \ C \\
\text{fail} \ C
\]

indexing instruction

miscellaneous basic instruction

The *rely_on_me_otherwise* instruction specifies the label L of the code to which to branch upon the exception C being handled. It encodes an exceptional choice point specified by an otherwise/2 predicate. When it is executed, a choice point is created by saving registers A2 and A1, the current environment pointer, the current continuation pointer, the prior choice point pointer, the address of the next clause, the current trail pointer, the current heap pointer, the current stack pointer and the prior exceptional choice point pointer. The heap and stack top registers are updated, and the exceptional backtrack register is set to point to the new exceptional choice point. The last choice point register is left unaltered.

The *fail C* operation initiates exceptional backtracking to the most recent prior exceptional choice point given by the exceptional backtrack register. The trail is "unwound" as far as is indicated by the trail pointer in the last exceptional choice point indicated in the exceptional backtrack register. References are popped off the trail and the variables they address are reset to unbound. The exceptional backtrack register is also updated from the exceptional choice point’s prior exceptional choice point field. If the exceptional choice point’s first argument is not the wild card *any*, a variable or the constant *C*, the backtracking process is continued from the prior exceptional choice point and so on. Where *C* itself is the wild card *any* or a variable, it matches with the first exceptional choice point name and in the case of a variable binds the variable to that name. Upon a successful match the machine registers are restored from the choice point, and the program pointer is set to the alternative clause pointer value in the choice point. Where the more expressive variant that allows escape exceptions to pass back atomic values is employed, a second extra field in the choice point would store the argument to be unified with the escape exception’s atomic value. This unification would be performed alongside exception name matching [Michaelson & Gregory 1990].
An example illustrates how the first instruction is used. The following two Prolog clauses

```prolog
persist(G, Result) :- otherwise(any, persist(G, Result)),
  call(G),
  yes = Result.
persist(G, no).
```

would be encoded by WAM instructions as follows

```wam
persist/2: try_me_else C2
  rely_on_me_otherwise C1, any % branch here on any exception
  call call/1, 0
  put_constant yes, A1
  execute /=2

C1: trust_me_else fail
  execute persist/2

C2: trust_me_else fail
  get_constant no, A2
  proceed
```

The `fail/1` call is translated into the `fail C` instruction. This instruction is also executed by the emulator for built in exceptional circumstances requiring recovery.

Exceptional choice points create no special memory management problems. They are just one cell longer (two if atomic values can be passed back with the escape exception) than ordinary choice points. As ordinary backtracking unwinds the local stack, exceptional choice points above the last ordinary choice point get popped off the local stack as a side effect of peeling back the local stack. Similarly as ordinary choice points get pruned by the execution of cuts, exceptional choice points above the point of pruning get popped. This is exactly as it should be. Exceptional choice points persist only so long as their part of the evaluation tree is active. Apart from updates to the exceptional backtrack register, their presence is invisible to the normal operation of the Prolog engine.

**Conclusion**

The exceptional choice points mechanism is a modest extension to Prolog. It can be implemented by fairly small extensions to the WAM instruction set and to WAM emulators. It exploits the general nature of
backtracking, which is already available in Prolog systems, to deliver escape exception, recovery and error handling. Prolog applications would benefit from having access to an exception handling facility based upon backward recovery to improve their properties of fault avoidance and fault tolerance [Cristian 1982]. It would complement notify exception handling and would augment other methods of ensuring robustness for Prolog systems. Escape exception handling can be understood both procedurally as a recovery control feature and declaratively in terms of the qualification which an exceptional alternative makes to a set of clauses. Thus escape exception handling preserves the duality of interpretations which is logic programming’s essence.
Chapter 7

7. Programming Coupled Resolution Engines

7.1. Introduction

This chapter explains how coupling together resolution engines executing Prolog and a CLP language as described in chapter three can be used to program multi-user knowledge based systems in logic. The Prolog engine is assumed to execute standard Prolog with extensions for delayed goal and escape exception handling. The CLP engine is assumed to execute the lingua franca described in chapter four. A prototype of much of the architecture has been developed in C to run under Unix. The prototype supports the CLP lingua franca, a full two way interface for translating full GHC and Parlog to this lingua franca and back again, and a full range of primitives. Details of the implementation are given in Appendix B. It can also invoke multiple Prolog computations using a C version of Prolog-X which runs under Unix. Prolog-X enables the prototype to exercise the facilities of a Prolog Database Machine [Wong & Williams 1988]. The prototype helps to show that the architecture can meet its abstract requirements by giving them concrete programming interpretations which can be executed.

7.2. Requirements of a Knowledge Based System

Chapters two and three elicit requirements for programming a knowledge based system in logic using a coupled Prolog-CLP system. To achieve this, the Prolog engine must support

- concurrent multi-threaded execution
- delayed goal handling
- escape exception handling
- standard and special purpose primitives

One way of securing concurrent multi-threaded execution of Prolog on a uni-processor without much interference in the design of the basic WAM engine is to exploit a Unix environment like Sun OS 4.0. Shared database Prologs invoked from within the same CLP meta-call would run as different lightweight processes with private stacks in a common Unix process with a single symbol table and code area. Private database Prologs invoked from within separate meta-calls would run as separate Unix processes with wholly private
memory. Fair concurrent execution would be achieved by the operating system. Delayed goal handling using async/2 can be supported along similar lines to implementing freeze/2 [Carlsson 1987], and a strategy for implementing escape exception handling in Prolog was sketched in chapter six. Some special purpose primitives are also required. The CLP engine must support

- control meta-call
- deep guards
- fair or programmable scheduling control
- standard and special purpose primitives

The CLP system can be realised by the lingua franca described in chapter four, and implemented along the lines sketched in chapter five. The chapter five implementation supports the control meta-call, and scheduling is both and-fair and programmable through the extended control meta-call. Some standard and special purpose primitives are also required.

Providing logic programming systems meeting these requirements exist, it should be possible to configure the implementation of each engine to achieve coupled execution along the lines explained in chapter three without much adaptation of either. The CLP engine will allow the use of the full range of systems programming techniques developed for the CLP languages [Shapiro 1984, Clark & Gregory 1984, Clark & Gregory 1986, Foster 1988] and the Prolog engine will allow the full range of techniques for programming knowledge based systems in Prolog [Clark & McCabe 1982, Takeuchi & Furukawa 1985, Sterling & Shapiro 1986 chapters 19, 21]. The benefits of combining both can then be realised along the lines to be described. The programs presuppose a coupled system executing the lingua franca and Prolog.

### 7.3. Meta-Interpreters

An important measure of the expressiveness of a logic programming language is its ability to express concisely an interpreter of the language in itself. Meta-interpreters of the lingua franca, of Prolog, and of the combined CLP-Prolog language using the four interfaces defined in chapter three are given below. They help illustrate the programming style of the lingua franca and the coupled language. They also justify the claim that the coupled approach is reasonably expressive.
7.3.1. A Lingua Franca Meta-Interpreter

A meta-interpreter of the lingua franca of GHC and Parlog can be concisely expressed as follows:

\[
\begin{align*}
\text{meta\_clp}((A, B)) & \leftarrow \text{true} \mid \text{meta\_clp}(A), \text{meta\_clp}(B) \\
\text{meta\_clp}(A) & \leftarrow \text{primitive}(A) \mid \text{call}(A, \text{yes}) \\
\text{meta\_clp}(A) & \leftarrow \text{true} \mid \text{clauses}(A, \text{Cls}), \text{reduce}(A, \text{Cls}, B), \text{meta\_clp}(B) \\
\text{reduce}(\text{Goal}, [(H:-G|B)|\text{Cls}], B1) & \leftarrow \text{match}(\text{Goal}, H, G) \mid B = B1 \\
\text{reduce}(\text{Goal}, [\text{_,_}]|\text{Cls}], B1) & \leftarrow \text{reduce}(\text{Goal}, \text{Cls}, B) \mid B = B1 \\
\text{reduce}(\text{Goal}, [((H:-G|B)|\_,_]|\text{Cls}], B1) & \leftarrow \text{match}(\text{Goal}, H, G) \mid B = B1 \\
\text{reduce}(\text{Goal}, [\text{_,C}]|\text{Cls}], B1) & \leftarrow \text{reduce}(\text{Goal}, [C]|\text{Cls}], B) \mid B = B1 \\
\text{match}(\text{Goal}, H, \text{Guard}) & \leftarrow H \leq \text{Goal} \mid \text{call}(\text{Guard}, \text{yes})
\end{align*}
\]

The meta-interpreter is invoked by

\[
\text{?- meta\_clp}(\text{Goals}).
\]

The first clause of \text{meta\_clp}/1 handles conjunctions, which are always executed in and-parallel. The second clause uses \text{primitive}/1 to recognise primitives. It evaluates them with the meta-call \text{call}/2, unifying its second argument to \text{yes} or \text{no} on the meta-call’s success or failure. The third clause of \text{meta\_clp}/1 handles resolution. Clauses retrieved in the second list argument of \text{clauses}/2 define the relation of the goal given in the first argument. These clauses have their selectability tested by \text{match}/3. The first clause of \text{reduce}/3 examines whether the leading clause can reduce the goal. The second clause searches the next clause in parallel. The third clause examines the last clause in a group of clauses to be searched in parallel. The sequential search operator after it ensures that the fourth clause is only considered after the first three have been considered and rejected. It continues clause search after all previous meta-interpreted clauses have been shown to be unable to reduce the goal. The single clause for \text{solve}/3 ensures that the head match is performed before the guard is called. The primitive \text{<=}/2 performs the one-way unification. It ensures that unifying its two arguments would not instantiate or share variables in its right hand argument.

This meta-interpreter is concise like Ueda’s GHC meta-interpreter [Ueda 1986a] 4.6.5. However, it handles primitives and sequential search as well. Mode declarations get in the way of expressing a concise Parlog meta-interpreter, but one can be written for the modeless language Kernel Parlog [Gregory 1987] p.64-69 instead. A meta-interpreter for modeless Flat Parlog in Parlog shows the idea [Foster 1988] Appendix I.
7.3.2. A Prolog Meta-Interpreter

Prolog’s aptness for writing meta-interpreters is well known [Sterling 1984]. A concise meta-interpreter for Prolog which handles primitives and the cut developed by the author is as follows:

```
meta_prolog(Goals) :- solve(Goals, Cut, V),
    ( V = true; ( V = fail, Cut = cut, !, fail ) ).

solve(!, cut, true).
solve(!, cut, fail) :- repeat.
solve((A,B), Cut, V) :- !, solve(A, Cut1, V1),
    ((Cut1 = cut, V1 = fail, repeat); solve(B, Cut2, V2) ),
    value(Cut1, V1, Cut2, V2, Cut, V).
solve(A, nocut, true) :- resolve(A).

resolve(G) :- primitive(G), !,
    call(G).
resolve(G) :- clause(G, Body),
    solve(Body, Cut, V),
    ( V = true; ( V = fail, Cut = cut, !, fail ) ).

value(cut, fail, _, _, cut, fail) :- !.
value(_, _, cut, V, cut, V) :- !.
value(Cut, V, _, _, Cut, V).
repeat :- true ; repeat.
```

The meta-interpreter is invoked by

```
| ?- meta_prolog(Goals).
```

This single argument call is expanded by two arguments to record the passing of cuts and whether Goals is true or fails. The first time a cut is encountered in the attempt to satisfy Goals, the first clause for solve/3 makes this a cut true combination. Subsequent backtracking makes the second clause turn this to a cut fail combination. These cut fail combinations are propagated across the goal level they originated on by the third clause for solve/3. Goals in the body after an earlier occurrence of a cut fail combination are ignored until the whole clause body has been traversed. The cut fail valuation cannot be undone by backtracking, once it has been created, because of the use of the endlessly resatisfiable goal repeat. The second clause for resolve/1 shows that if in the execution of the body of a clause, a cut fail combination is encountered, then the parent goal is made to fail. However, the parent goal resolve(G) does not pass back details on why the
goal failed. Thus the effect of a cut fail combination becomes simple goal failure at a higher level. value/6 combines the status regarding cuts and success or failure for goal conjunctions.

### 7.3.3. A Coupled CLP-Prolog Meta-Interpreter

A coupled CLP-Prolog meta-interpreter has to be written in both the CLP language and Prolog. The CLP side can be adapted from the lingua franca meta-interpreter given earlier as follows

```prolog
meta_clp((A, B)) :- true | meta_clp(A), meta_clp(B).
meta_clp(prolog(A)) :- data(A) | prolog(meta_prolog(A));
meta_clp(A) :- primitive(A) | call(A, yes);
meta_clp(A) :- clauses(A, [C|Cls]) | reduce(A, [C|Cls], B), meta_clp(B);
meta_clp(A) :- true | meta_prolog(A).
```

The meta-interpreter is invoked at the CLP side by

```prolog
| ?- meta_clp(G).
```

The new clauses for meta_clp/1 work like the old clauses except for the second and last clauses. The new second clause for meta_clp/1 handles the transiently communicating interface to Prolog prolog/1. It uses the primitive data/1 to delay execution until the argument A is instantiated. The new last clause handles the atomic interface to Prolog. The last clause for meta_clp/1 is invoked only if the goal argument of meta_clp/1 is neither a conjunction, a primitive, prolog(A) or user defined, which makes it by default a call over a definite clause relation. As the Prolog meta-interpreter meta_prolog/1 is only defined in the Prolog system, the body goal invokes the Prolog system atomically.

The Prolog side assumes that invocations of the transiently communicating interface clp/1, which occur in the body of a rule, are re-written in the canonical fashion used in chapter three. This consists of a single call to clp_call/2 at the start of the rule body and a call to data/1 at the end. Thus the clause

```prolog
relation(A, B, C) :- try(B, D),
          clp(test(A, D)),
          check(C, D).
```

is replaced by the clause
relation(A, B, C) :- clp_call(test(A,D), X),
try(B, D),
check(C, D),
data(X).

The meta-interpreter is adapted from the earlier meta-interpreter for Prolog by changing the clauses for
resolve/1 to the following

\[
\begin{align*}
\text{resolve}(G) & :\text{primitive}(G), !, \text{call}(G). \\
\text{resolve}(G) & :\text{clause}(G, Body), \\
& \text{body}(Body, B), \\
& \text{solve}(B, Cut, V), \\
& ( V = \text{true}; ( V = \text{fail}, Cut = \text{cut}, !, \text{fail} ) ). \\
\text{resolve}(G) & :\neg\text{clause}(G, _), \\
& \text{meta_clp}(G).
\end{align*}
\]

\[
\begin{align*}
\text{body}((\text{clp_call}(A,X), B), B) & :! , \text{clp_call}(A, X). \\
\text{body}(B, B).
\end{align*}
\]

The meta-interpreter is invoked at the Prolog side by

\[
| ?- \text{meta_prolog}(G).
\]

The old second clause for resolve/1 is replaced by a new clause. It uses body/2 to handle the transiently
communicating interface to the CLP computation. The new third clause handles the atomic interface. It is
only invoked if the argument to resolve/1 is neither a primitive nor a user defined goal. As the CLP meta-
interpreter is only defined in the CLP system, the meta_clp/1 goal invokes the CLP engine atomically.

The conciseness of this coupled meta-interpreter for the coupled system helps confirm how natural, simple
and expressive the four interfaces are. An added bonus is that the No Complex Term CLP Binding restric-
tion does not have to be separately enforced.

### 7.4. Concurrent Resolution

Concurrent knowledge based systems can be realised on the coupled architecture in many ways. Both the
CLP engine and the Prolog engine can set up interactive interfaces to Prolog for multiple users at the same
time. This shows the basic capacity for supporting a multi-user knowledge based system.
7.4.1. A Multi-User Prolog System

A simple multi-user Prolog system can be programmed on this system using the atomic interface to Prolog from the CLP engine as follows.

\[
\text{user(File) ::= open(File) | user(yes, File).}
\]

\[
\text{user(yes, File) ::= write(´yes\n?-, File) | query(R, File), user(R, File).}
\]

\[
\text{user(no, File) ::= write(´no\n?-, File) | query(R, File), user(R, File).}
\]

\[
\text{user(end, File) ::= true | write(halted, File).}
\]

\[
\text{query(R, File) ::= read(G, _, File) | execute(R, G);}
\]

\[
\text{query(R, File) ::= true | R = no.}
\]

\[
\text{execute(R, end_of_file) ::= true | R = end;}
\]

\[
\text{execute(R, G) ::= true | horn(R, G).}
\]

This program can be invoked for a terminal /dev/tty1 by the CLP call

\[
\text{\texttt{| ?- user(´/dev/tty1´).}}
\]

A prompt is issued and then a query is read in and parsed by the primitive read/3. The prototype supports read/3 as a primitive for the sake of performance, although it could obviously be written in a CLP language directly. Its action is to read in asynchronously a Prolog syntax term from File and to unify its first argument with the term. If read/3 encounters a syntax error, the second clause for query/2 reports the query result as no. If the parse succeeds, the goal horn(Result, G) defined in Prolog as follows is invoked

\[
\text{horn(yes, G) ::= call(G).}
\]

\[
\text{horn(no, G).}
\]

Multiple Prolog interfaces can be realised by executing user/1 goals concurrently for each terminal

\[
\text{\texttt{| ?- user(´/dev/tty1´), user(´/dev/tty2´), user(´/dev/tty3´).}}
\]

The fully asynchronous action of the read/3 primitive enables all these interfaces to be serviced concurrently. Because all the calls to Prolog are made within a single meta-call environment, the One Database per Meta-call rule makes them all operate over the same database state. To make them all operate over
separate database states, the calls to `user/1` merely have to be changed to

```prolog
?- user('/dev/tty1'), call(user('/dev/tty2'), _, _), call(user('/dev/tty3'), _, _).
```

The effect is then to invoke three different Prolog processes with their own private code areas. Clearly mixtures of Prologs having private databases and sharing them can be created by mixing these modes

```prolog
?- user('/dev/tty1'), call(user('/dev/tty2'), user('/dev/tty3'), _, _).
```

It is also possible to run these multiple Prolog interfaces from a Prolog computation. If the Prolog relation `multi_user/1` is defined as follows

```prolog
multi_user([File|Files]) :- clp(user(File)), multi_user(Files).
multi_user([]).
```

then the Prolog query

```prolog
?- multi_user(['/dev/tty1', '/dev/tty2', '/dev/tty3']).
```

will set up three concurrent interfaces to three terminals using the CLP engine. These simple interfaces show how more complex interfaces built up using low level input primitives like `get(Char, File)` can be invoked to serve as the access channels for multi-user knowledge based systems.

### 7.4.2. Controlling Resolution Engines

Parlog-style control meta-calls make it easy to suspend and resume particular interfaces using the facilities of the meta-call’s control stream [Gregory et al 1989]. The second argument of `call/3` is a list which can be bound to control instructions like `suspend`, `stop`, and `resume`. These control the activities of goals within the meta-call’s computation tree.

```prolog
?- call(test([A|B], C, _), C = [suspend|Cs], Cs = [resume|_]).
```

In this example the meta-call’s control stream is bound to `suspend` which suspends the meta-call’s execution and then execution is resumed by `resume`. Where a Prolog process is invoked from within the meta-call, it will be suspended and resumed as well. This is straight-forward to implement using Unix’s existing inter-process signaling mechanisms.
Control over Prolog computations from within a CLP computation can also be explicitly programmed using delayed goal and escape exception handling. Where a simple abort facility is wanted over a Prolog computation from its initiating CLP computation, the transiently communicating interface proclog/1 can be used in the following CLP call

\[
\text{prolog(abort(Goal, Control)).}
\]

If another goal in the CLP computation binds Control, the Prolog computation can be made to finish early. 

\text{abort/2} is defined in Prolog as follows

\[
\text{abort(Goal, Control) :- copy(Goal, G),}
\]
\[
\quad \text{freeze(Control, (Goal = G, halt)),}
\]
\[
\quad \text{call(G).}
\]

Instantiating Control wakes the frozen goal \text{Goal = G, halt} which reveals the current bindings for the goal being executed to the CLP computation and halts the Prolog computation. Another interface is the following

\[
\text{leash(Goal, Suspend, Resume) :- freeze(Suspend, data(Resume)),}
\]
\[
\quad \text{call(Goal).}
\]

If the Prolog goal \text{leash/3} is invoked within the CLP engine

\[
\text{prolog(leash(test(A,B), Suspend, Resume))}
\]
then binding Suspend in the CLP engine will result in the goal data(Resume) suspending further execution of the Prolog goal test(A,B) until the binding of Resume in the CLP engine restarts the Prolog computation.

A more complex form of control is necessary to stop the Prolog computation at selected points in its computation history. It can be realised as follows. An extra argument is added to relevant Prolog clauses and goals so that a clause such as

\[
\text{try(A, B) :- check(A, C), monitor(C, B).}
\]

is transformed to

\[
\text{try(X, A, B) :- check(A, C), monitor(X, C, B).}
\]
This transformation provides a common variable $X$ which is accessible to relevant clause body evaluations throughout the program. Periodic escape exception handlers for the exception *fallback* can then be set up at appropriate points, where the extra argument $X$ is visible in the environment of each handler for *fallback*. This is done by transforming a clause such as the one above to the clause

\[
\text{try}(X, A, B) :\text{ otherwise}(\text{fallback}, X = \text{stop}), \text{ check}(A, C), \text{ monitor}(X, C, B).
\]

If *stop/2* is defined as follows

\[
\text{stop}(\text{Goal}, \text{Control}) :\text{ copy}(\text{Goal}, G), \\
\text{ freeze}(X, (G = \text{Goal}, \text{halt})), \\
\text{ freeze}(\text{Control}, \text{fail}(\text{fallback})), \\
\text{ arg}(1, G, X), \\
\text{ call}(G).
\]

then the *Control* variable of the CLP goal

\[
\text{prolog}(\text{stop}(\text{try}(_, A, B), \text{Control}))
\]

can be used to stop Prolog execution of *try(_, A, B)* using these transformed Prolog clauses at *fallback* handler points. When *stop/2* is called, *try(X, A, B)* is copied into *try(X1, A1, B1)*. The goal

\[
\text{try}(X, A, B) = \text{try}(X1, A1, B1), \text{ halt}
\]

is frozen on the variable $X$, the goal *fail(fallback)* is frozen on the variable *Control*, and the variable $X$ is shared with the first argument of the goal i.e. $X1$. *try(X1, A1, B1)* is then executed. When *Control* is bound in the CLP engine, an escape exception *fallback* is raised in the Prolog computation. The escape exception causes the computation to fail and return to the last handler for *fallback*, which terminates the computation by waking the delayed goal $G = \text{Goal, halt}$. This makes the new bindings for *Goal* visible to the originating CLP computation and terminates the Prolog computation. The copying of the goal argument in *stop/2* is necessary to stop possible revocation of variable bindings which have already been communicated to the CLP computation during the exceptional backtracking. Escape exception handling is also useful for concurrency control over updates to a shared database. It can be realised along the lines explained in chapter six.
7.4.3. Multiple Solutions Predicates

Clark and Gregory have shown that it is possible to use destructive assignment to program eager and lazy all-solutions interfaces, set/3 and subset/3 [Clark & Gregory 1987], to Prolog from Parlog on a Parlog-Prolog coupled system using an interface similar to prolog/1. They try to mitigate their recourse to destructive assignment by arguing that it is only an implementation technique and can be hidden from the user. However, if it is hidden from the user, then the user cannot use the same technique to program variations on the multiple solutions predicate. The user is forced only to use set/3 and subset/3 to obtain multiple solutions. To avoid this weakness a different technique is advocated for supporting eager and lazy multiple solutions meta-calls from the CLP engine to Prolog. It can be more generally employed by users without embracing destructive assignment. An eager all-solutions predicate eager/3 like Parlog’s set/3 constructor can be defined in the lingua franca as follows

\[
eager(T, \text{Goal}, \text{Sols}) \leftarrow \text{gensym(Name)} \mid \text{all}(\text{Name}, T, \text{Goal}), \text{solutions}(\text{Name}, \text{Sols}).
\]

\[
solutions(\text{Name}, L) \leftarrow \text{element}(\text{Name}, A) \mid L = [A|M], \text{solutions}(\text{Name}, M);
\]

\[
solutions(\text{Name}, L) \leftarrow \text{true} \mid L = [].
\]

\[
\text{element}(\text{Name}, A) \leftarrow \text{defined}(\text{Name}, 2) \mid S =.. [\text{Name}, A, B], \text{retract}(S), B == \text{yes}.
\]

\[
\text{update_db}(\text{Name}, A, B) \leftarrow \text{true} \mid S =.. [\text{Name}, A, B], \text{assert}(S).
\]

\[
eager/3 \text{ obtains a unique symbol } \text{Name} \text{ from gensym/1 and uses the Prolog relation all/3 to compute solutions in a Prolog computation, collecting solutions using solutions/2. Solutions are passed from the Prolog system to the CLP engine’s database using the relation update_db/3 which appends them to the end of the definition of the relation Name/2. These solutions are progressively picked up from the database by element(Name,A). element/2 fails on the no labelled clause after the last solution has been picked up. The CLP primitive defined/2 suspends until the CLP database is defined for the given relation, and then the new solution is retracted. The primitives =../2 and ==/2 act like Prolog’s similar primitives. However, their success or failure does not depend upon the time they are called, because they suspend if their arguments are insufficiently instantiated for them to proceed. all/3 is defined in Prolog as}
\]
all(Name, T, G) :- copy((T,G), (T1,G1)), every(Name, T1, G1).

every(Name, T, G) :- call(G), update_db(Name, T, yes), fail.
every(Name, T, G) :- update_db(Name, _, no).

The predicate copy/2 ensures its arguments are distinct copies. The solution element template argument T and the goal argument G of eager/3 are copied to ensure that bindings, which will be revoked, are not passed to the CLP computation. Successive solutions are generated by backtracking. Each solution is passed to the CLP engine’s database using the atomic interface by making a call over the relation update_db/3 which is only defined in the CLP engine’s database.

Assuming that the file person contains the Prolog clauses

```
person(god).
person(X) :- rational(X), animal(X).
```

then this all-solutions predicate eager/3 can be used from the CLP engine as follows by going

```
| ?- prolog(consult(person)).
yes
| ?- eager(X, person(X), L).

X = _0
L = [god, socrates]
```

A lazy multiple solutions constructor lazy/3 based on Parlog’s lazy multiple solutions constructor subset/3, can be defined in the lingua franca. However, the No Complex Term CLP Bindings restriction means that the list of solutions L cannot be incrementally extended with variables for new solution elements in the CLP engine and these new list extensions passed transiently to the Prolog computation for adding new solutions through a prolog/1 interface. However, by adding a new kind of suspension primitive to Prolog defined( Relation, Arity), which causes the Prolog engine to suspend execution until the Relation of that Arity is defined by at least one clause in the Prolog engine’s clause database, it becomes possible to achieve the same effect. Requests for new solutions can be made indirectly by the CLP engine making additions to
the clause database of the Prolog engine. lazy/3 mimics subset/3’s functionality. The one solution call

\[ \text{?- lazy(X, person(X), [A]).} \]

binds A to god, whereas the request for more solutions than there is

\[ \text{?- lazy(X, person(X), [A, B, C]).} \]

binds A, B and C to god, socrates and end i.e. two solutions and the null solution marker. lazy/3 can be defined as follows:

\[
\text{lazy(T, G, L) :- gensym(Name) | some(Name, T, G), answers(Name, L).}
\]

\[
\text{answers(Name, [A|M]) :- ask_db(Name, yes), element(Name, A1) | A = A1, more(Name, M);}
\]

\[
\text{answers(Name, [A|M]) :- true | end([A|M]).}
\]

\[
\text{answers(Name, []) :- true | ask_db(Name, no).}
\]

\[
\text{end([A|M]) :- true | A = end, end(M).}
\]

\[
\text{end([]) :- true | true.}
\]

\[
\text{element(Name, A) :- defined(Name, 2) | S =.. [Name, A, B], retract(S), B == yes.}
\]

\[
\text{update_db(Name, A, B) :- true | S =.. [Name, A, B], assert(S).}
\]

where some/3 and ask_db/1 are defined in Prolog as

\[
\text{some(Name, T, G) :- otherwise(end, true), copy((T,G), (T1,G1)), each(Name, T1, G1).}
\]

\[
\text{each(Name, T, G) :- call(G), next(Name, T), fail.}
\]

\[
\text{each(Name, T, G) :- update_db(Name, __, no).}
\]

\[
\text{next(Name, T) :- defined(Name, 1), S =.. [Name, yes], retract(S), update_db(Name, T, yes), !.}
\]

\[
\text{next(Name, T) :- fail(end).}
\]

\[
\text{ask_db(Name, Status) :- S =.. [Name, Status], assert(S).}
\]

The lazy multiple solutions predicate lazy/3 uses the Prolog call some/3 to compute solutions and the CLP call answers/2 to ask for and collect solutions. answers/2 uses ask_db/2 to add an extra clause Name/1 with a yes argument to the same Prolog database seen by some/3 and receives back solutions using eager/3’s update_db/3 mechanism for passing solutions to the CLP engine’s database. end/1 lazily binds extra solution stream elements to end where no more solutions can be found. Solutions are found in the Prolog
engine by \textit{each/3}. \textit{next/2} uses \textit{defined/2} to pause Prolog execution until the next solution is wanted and then sends a solution to the CLP engine using the CLP call \textit{update_db/3}. The escape exception handler \textit{end} ensures that when the clause \textit{Name/1} with its argument bound to \textit{no} is encountered (the signal from the CLP engine that no more solutions are wanted), an \textit{end} escape exception is raised by the second clause for \textit{next/2}. The Prolog computation for \textit{some/3} then gracefully terminates on success.

7.5. A Prolog Database Machine

Conventional Prolog engines need special support to handle resolution on a large scale. This requires a mass storage strategy for handling large numbers of clauses, and a rapid retrieval strategy for resolution over a large number of clauses. The relative cheapness of a megabyte of disc space as against a megabyte of RAM means very large knowledge based applications in Prolog favour a disc based storage strategy. One notable exception is a recent decision support system developed by the FBI in USA for FBI operatives to handle information on \textit{Terrorism and Organised Crime} which runs under Quintus Prolog on a Sun-3 using 120 megabytes of RAM and just squeezes into 110 megabytes of that space after compilation [FBI 88]. Some fast commercial Prolog systems based on the Warren Abstract Machine [Warren 1983] support virtual memory for the code area allowing the clause database to be stored mostly on disc, and paged into RAM as and when needed using optimal caching strategies to minimise page faults. Because this can be slow without special augmentation, more special purpose approaches have also been advocated to accelerate clause retrieval from disc using database machine technology. Coupled approaches like ICOT’s DELTA machine [Murakami et al 1983] and ECRC’s EDUCE [Bocca 1986] use existing relational database technology and interface Prolog systems to relational databases. Other approaches aim to improve the indexing methods of clauses held on disc. Examples include the superimposed codeword indexing scheme of MU-Prolog [Ramamohanarao & Shepherd 1985], and the balance and nested grid files of KB-Prolog [Bocca et al 1989]. In as much as these systems build upon relatively conventional Prolog systems they are all compatible with the coupled architecture approach being advocated here.

The prototype of the coupled architecture has been developed for use with a multi-user Prolog Database Machine [Massey 1988]. It has been constructed to enable a specially adapted Prolog system to handle very large sets of clauses efficiently in an integrated fashion. Such a Prolog system stores some clauses in main memory and uses the Prolog Database Machine to store large numbers of clauses on disc. The
Database Machine uses special hardware and software to accelerate retrieval by head unification of goal matching clauses stored on disc. Multiple logic programming systems are allowed concurrent access to these clauses. Apart from access constraints caused by concurrent use of the same set of clauses, access to all clauses is transparent to a connected Prolog system. Details of the Prolog Database Machine and how it is programmed will be briefly examined. They show how a coupled architecture like the one being advocated here can be integrated with other kinds of development work on Prolog to enable the coupled architecture to handle resolution on a large scale.

Two large knowledge based system applications have been developed for the Prolog Database Machine. A logic database system written in Prolog uses the extended clause handling facilities of the Prolog Database Machine to manage deduction over significant amounts of knowledge [Williams et al 1988]. An expert systems interface in turn uses the facilities of the logic database to handle access to large sets of expert system rules [Salvini 1989].

7.5.1. Database Machine Components

The three main components of the Prolog Database Machine are

- **Prolog-X** specially adapted Prolog system
- **Clause Retrieval Server** software server for fetching clauses for Prolog-X
- **Clause Retrieval Hardware** hardware filter for fetching clause unifiers from disc

The Prolog system developed for use with the Prolog Database Machine is Prolog-X, an abstract machine based implementation of Prolog developed by W.F.Clocksin [Clocksin 1985] which supports standard Prolog with an extension for handling modules. Prolog-X is compiled to an abstract machine code which is emulated in software. Prolog-X has been specially adapted to make its clause retrieval requests over clauses stored on disc to an independent clause retrieval server [Massey et al 1989].

By separating the functionality of clause retrieval from a logic programming system and embedding it in a quasi-autonomous server, it becomes possible to provide a standardised interface to it. This interface can then be used to handle clause retrieval for several different clients. It also becomes possible to configure the Clause Retrieval Server [Massey et al 1989]. to support
concurrent access by multiple clients

distributed operation

knowledge base sharing

persistent clause storage

transaction based access arbitration

One server can support clause retrieval requests from several different clients concurrently using request scheduling with a single thread of control. Furthermore there is no requirement for the Clause Retrieval Server to reside on the same machine as its clients. Access across a Local Area Network can be provided almost as easily. Delocalisation makes it worthwhile to allow the Clause Retrieval Server to split up its retrieval responsibilities into a number of servers connected across the network, each able to handle retrieval requests by clients and by other servers. Thus retrieval requests, which are not able to be handled directly on one server, can be passed on to the relevant server for that relation elsewhere. Because the server stores clauses on disc and organises full access to them, the Prolog clauses held under its charge need not be reconsulted every Prolog session. Once these clauses have been put under the clause retrieval server’s charge, they can be accessed only by making visible relevant relations. By giving clauses to a clause retrieval server in this way, client Prolog systems are able to function as persistent Prolog systems. Multiple clients interrogating a single server may want common access to a single set of clauses. This creates no difficulty where only retrievals are concerned. However, where clients want to update shared sets of clauses, it becomes important to support locking and access arbitration so as to maintain consistency in clients’ views of shared sets of clauses. The clause retrieval server is given full control over concurrency and clause update management using a relation level locking scheme.

Each clause retrieval server is able to manage a software search for clause satisfiers of a goal over clauses stored on disc. A variety of indexing methods are employed to accelerate this. However, the main use of clause retrieval servers is to serve as software means of invoking specialised clause retrieval hardware for accelerating clause retrieval from disc. ICL’s Content Addressable File Store is one possible hardware engine for performing this task. A more specialised device is the CLARE engine [Wong & Williams 1988]. The main features of the CLARE clause retrieval hardware are

- VME bus based co-processor searching on-the-fly
- two stage filter 1) superimposed codeword indexing
2) partial test unification

A backend co-processor performs its search on-the-fly as clause data streams off disc. Two filters are employed for recognising clauses whose heads unify with a call. Either filter can be used independently or both can be employed in the given order. The first filter employs superimposed codeword plus mask bits matching [Ramamohanarao & Shepherd 1985]. A superimposed codeword is associated with each clause and matched codewords cause their corresponding clause to be retrieved directly from disc. The second filter performs tests directly on clause data to see whether unification between the goal and a clause head is possible. Both filters yield false positives but neither filter yields false negatives. These false positives are eliminated by the clause retrieval server before it passes retrievals back to its client. The filters are designed for use with a VME bus. Each fills a standard VME board.
One possible configuration of client logic programming systems, clause retrieval servers and hardware is shown by Figure 7.1. Multiple Prolog-Xs and a CLP engine, are using a network of clause retrieval servers linked to several pieces of hardware. The whole forms a distributed multi-user knowledge base machine.

### 7.5.2. Use of the Database Machine

Prolog-X clauses can either be held in internal modules in main memory or be held in external modules on disc under the control of the Prolog Database Machine. External modules appear almost like internal modules, except that all uses of external modules must be done within a transaction. A Prolog-X call

```
| ?- crs_loadmodule(db,’/tmp/db’).
```

opens a module called `db` and associates the persistent file `/tmp/db` with it. If the file already exists, then the clauses already in it become visible in the current module. The call

```
| ?- transaction(db_query(L)).
```

tries to satisfy `db_query(L)` over the clauses in the module `db` i.e. those in the file `/tmp/db`. Should concurrent access to the same file by another user cause transaction failure, the whole transaction meta-call fails without updates to clauses in the module during the transaction being committed to the external file. Successful transactions atomically commit their updates to the file on exit. Escape exception handling is used to ensure that failed transactions backtrack all the way to the beginning of a transaction in the fashion described in chapter six. In all other ways Prolog-X behaves like a conventional Prolog engine. Thus the Prolog Database Machine radically extends the scale of the clause database Prolog-X can handle with minimal change to the user’s model of programming over it.

### 7.6. Conclusion

This chapter has demonstrated how to program in logic multi-user knowledge based systems on coupled resolution engines which are multi-processed. Fine grained concurrency is achieved by and-parallel execution within the CLP engine and among multiple Prologs and the CLP engine by the host operating system. On a Unix based multi-processor like the Sequent series these separate Prolog processes can execute on separate processors. They can be coordinated by either a single processor implementation of the CLP
engine or a distributed or tightly coupled parallel implementation. The coupling structure of the architecture coordinates the separate components of the knowledge based system in a logic programming framework. A prototype of the system has been developed using Prolog-X and that has given the prototype access to resolution over a large number of clauses by using the facilities of a Prolog Database Machine. This capability establishes the compatibility of a coupled resolution engine approach to logic programming with scale expanding approaches using advanced indexing techniques and knowledge base machines. The next generation of computer systems will be knowledge processing engines, and the approach described here elucidates one way to realise them by executing coupled CLP and Prolog programs in parallel.
Chapter 8

8. Conclusions

8.1. Summary of Argument

This chapter summarises the argument of this dissertation, discusses the limitations of coupled resolution, and mentions possible further avenues for this research. It also comments on the relation of a coupled resolution engine approach to the Japanese FGCS project. The argument of this dissertation is that coupling a concurrent logic programming engine with a multi-threaded Prolog engine is a solution to the problem of how to program concurrent knowledge based systems in logic for multi-processing. Its reasoning can be recapitulated as follows.

8.2. Knowledge Processing Engines

Research on a fifth generation of computer systems aims at developing high performance knowledge processing engines based on logic programming. Logic programming has been selected as the computational paradigm for the next generation of computers for several reasons. It is able to automate deduction, which is essential to the extraction of knowledge from knowledge. It also has very considerable potential for parallel execution, which is needed for boosting the performance of software and for exploiting the parallelism of the next generation of computer hardware. In addition software development productivity can be boosted by articulation in logic, by giving programs meanings with ready natural language readings and by making programs amenable to examination by formal methods. Practical logic programming languages have already demonstrated the aptness of the underlying computational paradigm for developing knowledge based systems. However, difficulties remain with the methods being developed for using logic programming languages to program knowledge based systems. So far these methods make it problematic for a logic programming system to combine adequate deductive capabilities over knowledge bases with the ability to support systems programming, while remaining suitable for parallel execution. This combination is important for programming both the systems and the deductive aspects of knowledge based systems in logic. Addressing this problem has been the subject of this dissertation’s research.
8.2.1. Programming Knowledge Bases in Logic

The context of the proposed solution to this problem was set by elucidating key concepts used in this research - logic, logic programming, parallel execution of logic programs, knowledge and knowledge based systems. They were characterised and related together, and the way two main types of logic programming language, Prolog and the CLP languages, relate to a proof procedure for a fragment of first order logic was brought out. Two important properties, exhibiting don’t know non-determinism in resolution and being able to support the process interpretation of systems programming, were shown to be exhibited respectively by Prolog and the CLP languages.

The requirements on logic programming languages to support knowledge based systems were then elicited. It was argued that a resolution based parallel logic programming system must support both don’t know non-determinism in resolution and the process interpretation of systems programming in order to realise both the deduction and systems aspects of knowledge based systems. Knowledge based systems must be able to perform a complete and terminating search of their knowledge to sustain an adequate deductive querying capability. Support for don’t know non-determinism in resolution, as exhibited by canonical, and-parallel and or-parallel Prologs, is needed for doing this effectively. The systems programming requirements of knowledge based systems must also be supported. That requires the ability to handle the synchronisation and concurrent execution of multiple tasks on a fair basis, which suitable implementations of the CLP languages are able to handle. However, Prolog is unable to support the process interpretation of systems programming and the CLP languages are unable to support don’t know non-determinism in resolution.

8.2.2. Resolving Deduction with Systems in Logic

It was shown that the reason why these languages did not support both characteristics was because implementations of each characteristic work against implementations of the other characteristic. Don’t know non-determinism is sustained by giving a trial binding character to variable instantiation in the execution of Prolog and by sequential resolution in canonical uni-processor implementations. However, making bindings revocable undermines the process interpretation of systems programming by interfering with the irrevocability of message passing between system components using variable bindings. Furthermore a sequential process of resolution makes it hard to model concurrency among components of systems. The limitations of this sequentiality are not circumvented by multi-processing or-parallel or and-parallel Prologs, because
parallel Prologs lack adequate synchronisation constraints to model dynamically interacting systems. Co-routining Prologs mitigate the failings of Prolog to synchronise concurrent processing of tasks, but fail to support fair concurrency. Thus while Prolog is apt for supporting a deductive querying capability, neither it nor any of its parallel or co-routining variants is suitable for systems programming. The reverse side of the coin is that the CLP languages can support systems programming applications quite well. This capability arises from their use of committed bindings, concurrent committed choice resolution, general synchronisation constraints, and their ability to be implemented to support fair concurrency. However, unlike Prolog and its variants, their committed clause choice and committed variable binding mechanisms preclude don’t know non-determinism. This renders them unable to support a deductive querying capability directly.

Several different ways of reconciling don’t know non-determinism with the process interpretation of systems programming, while retaining concurrency suitable for parallel execution, were reviewed and rejected. The most plausible alternatives, Parallel NU-Prolog and Andorra Prolog, combined and-parallelism and goal delay mechanisms, but failed to sustain fair concurrency or sufficiently expressive committed choice resolution. One viable alternative remained - coupling together CLP and Prolog engines, so that CLP and Prolog computations could execute concurrently. Deductive querying parts of a multi-user knowledge based system could be realised by trial binding computations on a multi-threaded Prolog engine. Systems programming parts could be realised by committed binding computations on a CLP resolution engine. Interactions among these computations could be sustained in a multi-processing environment by message passing interfaces to realise all the required capabilities together.

8.2.3. Coupled Resolution Engines

Clark and Gregory first advocated a hybrid scheme called Parlog and Prolog United for executing the CLP language, Parlog, and Prolog together. However, their proposal undermines the process interpretation of CLP execution by introducing revocability to messages passed by variable bindings. Their proposal also compromises efficient multi-processing by connecting Prolog and Parlog engines using shared memory, and mars efficient implementation of a CLP engine by introducing into it the complex machinery for sustaining chronological backtracking. A different approach is advocated instead. A CLP engine is coupled with a multi-threaded Prolog engine. Communication among the multiple computations is realised by message passing. This enables the architecture to be multi-processed without significant contention for shared
memory among different computations. Bindings to shared variables, once communicated between computations, are made irrevocable. This avoids having to burden the CLP engine with chronological backtracking machinery, retains irrevocability for CLP bindings, and ensures that the process interpretation of systems programming can still be applied to CLP execution.

Four interfaces between Prolog and CLP computations were defined, to allow coupled resolution engines to combine effectively Prolog’s ability to handle deduction over knowledge bases with a CLP language’s ability to program concurrent systems. Concurrency among mutually invoking Prolog and CLP computations is sustained partly through and-parallel execution of CLP computations on the CLP engine, and partly by executing multiple Prolog processes and a CLP engine under an operating system like Unix. The design of suitable coupling interfaces and high level methods for realising these forms of coupling were given. These coupling methods presupposed delayed goal handling capabilities in Prolog along the lines of Prolog-II’s \texttt{freeze/2} primitive and the ability to handle escape exceptions in Prolog.

\section*{8.2.4. CLP Language and Engine Mechanics}

There are various CLP languages which could be used to realise the CLP engine of a coupled resolution engine system. Contrasts and comparisons were drawn between their respective merits. A sufficiently expressive yet tractably implementable CLP language was sought that could be the basis for the CLP resolution engine part of the architecture being proposed in this dissertation. It was argued that two of the most important concurrent logic programming languages, GHC and Parlog, looked likely candidates for this role. However, they balance the requirements of clean semantics and good control rather differently, and their respective characteristics were distinguished from each other. Since concurrent logic programming would benefit from both, a lingua franca of these languages was characterised. It enables a single implementation of the lingua franca to support both languages transparently. By supporting both languages it was argued that the lingua franca can provide a user with the semantically desirable properties of GHC, the control capabilities of Parlog and any performance benefits its simplicity might bring on a single implementation.

Aspects of the design of an emulator for supporting this common CLP language were then considered. A method for economising on the use of persistent data storage areas for shared variables was proposed using a delayed process promotion scheme. A method was also proposed for adapting lazy process scheduling to handle sequential clause search. Schemes for handling meta-calls and process suspension were also
described. These implementation issues helped to clarify the space of implementation possibilities in realising the CLP engine. This helps to redress the balance of implementation considerations from well-defined Prolog implementation schemes to more open choice among CLP implementation schemes.

8.2.5. Prolog Exceptions

While neither Prolog nor its variants can serve as systems programming languages in their own right, Prolog needs extensions for handling interfaces with concurrent computations. This requires delayed goal and escape exception handling facilities. Delayed goal handling facilities are well understood, but escape exception handling is also needed. It was argued that giving Prolog escape exception handling would also benefit Prolog in making it more robust and fault tolerant. A scheme for escape exception handling in Prolog, which exploits the general nature of backtracking that is already available in Prolog systems, was advanced and justified. It was argued that escape exception handling can be understood both procedurally as a recovery control feature and declaratively in terms of the qualification which an exceptional alternative makes to a set of clauses. It preserves the duality of interpretations which is logic programming’s essence. Extensions to the canonical approach to implementing Prolog on uni-processors to enable it to support the exceptional choice points used by escape exceptions were described. They showed how an exceptional choice points mechanism is a modest extension to Prolog, which can be implemented by fairly small extensions to the WAM instruction set and to WAM emulators.

8.2.6. Coupled Programs

The argument of this dissertation is that a coupled resolution engine architecture is a feasible solution to the problem of how to combine in a logic programming framework the systems and deduction aspects of knowledge based systems, while retaining adequate scope for parallel execution. This claim was redeemed by showing how the coupled architecture could be programmed. First of all the simplicity and expressive power of the four interfaces was vindicated by giving a concise and economical Prolog-CLP program which is a complete coupled meta-interpreter of the main features of each resolution engine through all four interfaces. Simple programs were then given that showed how to program multiple concurrent interfaces to Prolog engines within the framework of using a coupled program. This established the capacity to support concurrent knowledge based systems - using Prolog programs to execute queries over the knowledge base,
and CLP programs to sustain concurrency among them. Varieties of ways to manage and control Prolog computations from CLP programs were also described. Programs to realise both eager and lazy multiple solutions predicates were also given to demonstrate further that coupled programs allowed systems programming control to be harnessed with don’t know non-deterministic search in the ways needed to support concurrent knowledge based systems in logic.

8.2.7. A Practical Approach

The proposed solution to resolving deduction, systems control and parallel execution in a logic programming framework uses Unix and existing implementation technology for Prolog and CLP implementations with some minor adaptations. It could be realised on a uni-processor like a Sun-3 using Sun’s version of Unix OS 4.0. A state of the art WAM engine like SICStus Prolog could be interfaced with a CLP engine like SPM [Gregory et al 1989], with extra code for specialised primitives and extensions to the WAM engine for escape exception and delayed goal handling. The coupled system could also be realised under a multi-processor version of Unix like Dynix on a shared multi-processor like the Sequent series [Babb 1988] using SICStus Prolog extended by escape exception handling and an adapted version of a multi-processor implementation of Parlog like the JAM [Crammond 1988]. Thus it represents a practical approach which can build on existing implementation technology. The approach enables separate components of the macrostructure of a multi-tasking knowledge based system to be mapped onto separate computation threads, which are then concurrently processed. Thus the approach helps to match the large scale grain of the application’s task concurrency with a set of processes which can be executed at the same time. It does this in a way which makes it relatively simple and flexible to program.

8.3. Limitations of Coupled Resolution

Coupled resolution has general limitations as a way forward in parallel logic programming. The particular approach to coupled resolution also has limitations compared with other approaches like Clark and Gregory’s tightly coupled approach to coupled resolution. Table 8.1 summarises the properties of languages which attempt to support both stream and-parallelism and don’t know non-determinism.
Table 8.1 Languages for Programming Systems and Search Problems in Logic

It briefly contrasts some of the characteristics of rival approaches to reconciling search capabilities with and-parallelism in logic programming which were discussed in detail in chapters two and three. The loosely coupled approach to resolution advocated in this dissertation is labelled CLP-Prolog.

Coupled resolution has some general limitations as a way to reconcile parallel execution, stream and-parallelism and don’t know non-determinism. Besides its failure to extract much potential parallelism in such applications, which is discussed later, there are two other important limitations.

- the solution provided is a hybrid rather than an integrated one
- its multi-threaded Prolog concurrency is coarse grained

Coupled resolution is at heart a hybrid solution. It seeks to reconcile diverse types of resolution by interfacing inference engines which support each type of resolution. This eclecticism lacks the integrated harmony which one would expect of a true synthesis of parallelism, systems programming capabilities and the kind of resolution proof procedure needed for knowledge based applications. Coupled resolution rather manifestly fails to transcend its heterogeneous origins by preserving unamended the programming styles appropriate to each component resolution engine. One might expect an adequate integrated solution to introduce a new programming style in place of former approaches to writing logic programs. This might be expected to enhance the declarative readings of logic programs. Andorra Prolog does rather better than coupled resolution approaches in this respect.
Developing an integrated language solution would seem to have been a major part of the motivation why Gregory has tried to go beyond coupled resolution in creating a language Pandora which aspires to integrate stream and-parallelism and don’t know non-determinism more closely [Bahgat & Gregory 1988]. Lack of integration in coupled resolution undermines coupled resolution’s claim to be an ultimately satisfactory solution intellectually. However, even a language like Pandora does not dissolve away all heterogeneity between the contending demands of stream and-parallelism and don’t know non-determinism. Pandora preserves that heterogeneity by substituting distinct phases of Parlog-like execution with don’t know non-determinism for coupled stream and-parallel and don’t know non-deterministic computations. It remains to be seen whether Pandora can be implemented in a memory efficient fashion.

A second general limitation of coupled resolution is that the style of concurrency achieved among don’t know non-deterministic searches is basically coarse grained. Coupled resolution shares this characteristic with Parallel NU-Prolog. Because Prolog computations have significant overheads in start up times and in their static memory requirements, it is not practical for coupled resolution systems to create or have in existence large numbers of Prolog computations at the same time. Nor is it efficient to incur the communications overheads with a Prolog computation only for trivial calls to Prolog. These limitations can be recognised by programmers, and programs written to minimise their effects. However, it is not ultimately satisfactory that implementation and efficiency issues should influence the programming style that much.

Specific limitations of the loosely coupled approach to coupled resolution compared with other approaches are that

- it lacks constraint handling capabilities
- its message passing approach has significant overheads

The proposal for loosely coupled resolution lacks the constraint handling capabilities of Parlog and Prolog United or PPU given to PPU by its lazy don’t know non-deterministic interface from Prolog to Parlog. PPU possesses this capability in common with Pandora and Andorra Prolog. This is a significant limitation of loosely coupled resolution compared with other approaches. The ability to handle and propagate constraints is emerging as a significant new dimension in logic programming [Jaffar & Lassez 1987, Jaffar & Michalov 1987], and a means for doing this declaratively in parallel would enhance the expressive and problem solving power of logic programming in a major way. Delayed goal handling capabilities using Prolog’s freeze/2 primitive [Carlsson 1987] can already be used to support constraint handling capabilities on top of Prolog
and loosely coupled resolution has that capability accessible to it already in its Prolog engine in much the same fashion as it is available to Parallel NU-Prolog. However, Prolog’s inherent sequentiality hampers loosely coupled resolution from exploiting that capability in parallel. In addition to providing constraint handling capabilities lazy don’t know non-determinism looks rather useful for AI type search applications in helping to contain explosive expansions of the search space. However, that greater expressive power cannot escape being justified in efficiency terms. In PPU’s case constraint handling capabilities require support for chronological backtracking in Parlog, which is problematic to implement efficiently.

PPU uses shared memory to store variables shared between Parlog and Prolog computations and allows both resolution engines to bind variables in each others heaps. This is relatively straight-forward to implement and avoids imposing significant restrictions like the No Complex Term CLP Bindings restriction on shared variables imposed by loosely coupled resolution. It also avoids imposing overheads associated with message passing upon each binding to a shared variable. These drawbacks have to be set against the clear advantages of loosely coupled resolution over tightly coupled resolution for multi-processing.

8.4. Further Research

This research has advocated multi-processing coupled resolution engines to realise concurrent knowledge based systems in logic. It would be interesting to pursue further several issues related to this research.

8.4.1. A Full Prototype

So far only a prototype of the atomic interface from a CLP implementation to Prolog has been implemented. A user’s view of it is described in appendix B. The prototype is written in C, and, although relatively customised, only interprets the CLP language at a modest rate [Taylor 1987]. The prototype invokes a C version of Prolog-X with a specialised top level written in Prolog. It allows multiple Prolog computations to be initiated from a CLP computation and executed concurrently under Unix, but only with separate clause databases in each Prolog process. At the moment the prototype only realises multi-threaded execution over a shared Prolog database by one of two means. Either concurrent access is achieved by using the facilities of the Prolog Database Machine described in chapter seven from within Prolog-X. Or access is achieved non-concurrently by delaying subsequent calls to Prolog from within the same CLP meta-call until earlier ones over the same Prolog database have finished executing.
It would be interesting if a full prototype of the coupled system, which was able to use all four interfaces and could execute multiple Prolog computation threads over a shared database state, could be developed. It would then be possible to refine the methods of chapter three for realising the four interfaces and to develop practical experience in the art of programming coupled resolution engines. A working full prototype would make it rather easier to explore concurrency control problems in coupled execution like livelock and deadlock. A full prototype could also be instrumented and performance issues for coupled execution could be addressed.

8.4.2. Extracting More Parallelism

Multi-processing coupled resolution engines executing Prolog and a CLP language using message passing provides a loosely coupled processing framework for executing knowledge based systems in logic. However, it does not by itself extract and exploit all available parallelism within the application. Each CLP and Prolog computation thread is itself amenable to being multi-processed. The most realisable and useful possibilities would seem to be that the CLP engine could be multi-processed using local and shared memory [Crammond 1988, Sato et al 1987]. Each Prolog computation thread could also be executed in or-parallel using local and shared memory [Warren 1988]. The loosely coupled processing framework would realise concurrent execution of the various Prolog computations with the CLP engine’s execution.

In terms of a multi-user knowledge based system application, it would enable concurrency in the macro-structure of the application, arising out of multiple users initiating multiple concurrent tasks, to be exploited by multi-processing. Secondly, parallelism in the micro-structure of each knowledge processing task could also be exploited by using or-parallel execution to speed up don’t know non-deterministic search in finding all solutions to a knowledge base query. Lastly parallelism in the micro-structure of multiple interacting systems programming tasks and in concurrent execution at the CLP engine end of calls to and from Prolog computations could be exploited. This would be realised by using and-parallel execution to accelerate the processing of each part of the CLP computation. Both the macro-structure and the micro-structure would be coordinated in the single framework of an operating system suitable for multi-processing like Unix.
8.4.3. Beyond Unix

Unix only provides within the proposed scheme a currently available operating system framework for realising

- fair concurrent execution of various computations
- interactions with the file system
- message passing among various computations
- shared memory concurrency controls
- user interfaces

Plainly these services might be realised by other means. Handling interactions with the file system and user interfaces are normal operating system responsibilities, and are probably best left to operating systems, but the realisation of fair concurrent execution of a multi-threaded computation, message passing among these computations, and control over concurrent access by some of these computations to a shared code area could be realised more directly without using a general purpose modern operating system. In this way a less general, faster and more customised realisation of a multi-threaded Prolog and CLP coupled system could be attained.

8.5. Coupled Resolution and the FGCS Project

The Japanese Fifth Generation Computer Systems project is the centre piece of FGCS research activities. Of all the research going on in the world into parallel logic programming, it is the most sustained and integrated piece of research and development which aims at producing a fifth generation knowledge processing machine. It represents an impressive piece of organised, long term, innovative research, and is broadly on target in its central objectives. The record of the previous seven years research suggest that it will be seen through to a successful conclusion in 1992. Quite how successful it will be, depends a lot on the price at which Japanese manufacturers will be in a position to sell fifth generation computers. Some remarks on the FGCS project will be used to put this research into perspective.

The FGCS project is now entering its last phase of development, when its research strands are being brought together into a prototype of a fifth generation machine. In the last phase of the FGCS project the parallel knowledge base sub-system and the parallel inference sub-system will be merged to form an
integrated parallel machine which executes a variant of flat GHC called KL1. The design of the parallel inference machine is now relatively clear cut [Sato et al 1987], and a prototype of the parallel inference machine, the PIM/p, is being fabricated in VLSI. It will consist of 16 clusters of 8 processors. Target performance is 200-500 KLIPS for each processor and 10-20 MLIPS for the whole machine. It is due to be superceded by a later version of the PIM, embodying around 1000 processors, which will also be fabricated in VLSI. While progress with the PIM is far advanced, the design of the parallel knowledge base sub-system is still not finalised.

Because current plans for the parallel knowledge base sub-system still seem to be in the melting pot, a commentary on its future direction can only be speculative. Four pieces of research, the CHI, PHI and Mu-X systems and work on a parallel interface to knowledge bases from GHC, have formed the main strands of knowledge based systems research at ICOT during the FGCS project’s intermediate stage [Itoh et al 1988]. The CHI system is a large shared memory multi-Prolog system which is multi-processed. It uses both dynamic local and static shared clause databases by means of multiple-multiple name spaces. The PHI system is a distributed deductive database system using ICOT’s LAN in order to connect a number of PSI machines. The PSI machines execute Prolog and use attached hardware to perform relational operations over function free unit clauses held on disc. Mu-X is a multi-processor backend for PSIs, exercised by an adapted Prolog language. Mu-X executes term-relational queries (extended relational queries which replace equality checking with term unification) over full unit clauses held on large scale multi-port page memory [Yokota & Itch 1986]. The parallel interface to GHC is a special predicate \texttt{rbu/1} which handles a stream of term-relational algebra commands for retrieving and updating knowledge via a dedicated term-relational knowledge base architecture.

However, the indications are [Itoh et al 1988] that the parallel knowledge base sub-system will be mainly based upon enhancements of the Mu-X approach and the \texttt{rbu/1} interface. This suggests that it will be realised in its FGHC aspects on the same machine as the parallel inference sub-system. It will use the special predicate \texttt{rbu/1} to exercise from FGHC a dedicated system for operating a knowledge base by means of retrieval by unification operations. Furthermore the dedicated knowledge base sub-system will use large scale shared memory, interconnection and switching networks for hardware communications, and specialised unification, backtracking and indexing machinery to realise clause storage and retrieval in the parallel knowledge base machine. RBU retrievals initiated by successive commands on an \texttt{rbu/1} stream will be
compiled into lower level operations, which will be farmed out to other processing elements to execute in parallel over clause storage. Multiple solutions will be delivered incrementally as a list to *unification restriction stream or unification join stream* commands. Thus the one element stream *rbu/1* goal

```
| ?- rbu([urs(teacher, [1], student(female,x(1)), [1], X))].
```

\[ X = [\text{student(female,science(x(2))}, \text{student(female,arts(history)), ...}] \]

contains a unification restriction stream operation *urs/5*. It initiates a search of the first attribute of the term relation *teacher* for terms unifiable with the condition *student(female,x(1))*. The first attribute is derived as a result. Results are returned as a stream bound to the variable \( X \). An occurrence of \( x(N) \) where \( N \) is an integer signifies a variable. The one element stream *rbu/1* goal

```
| ?- rbu([ujs(teacher, [2], employee, [1], [3], X))].
```

\[ X = [\text{salary(5000,part_time), salary(12300,fulltime), ...}] \]

contains a unification join stream operation *ujs/5*. It is used to derive the third attribute of a result operation generated by a search of the second attribute of the term relation *teacher* and the first attribute of the term relation *employee* for unifiable terms. Results are returned as a stream bound to the variable \( X \).

Thus the FGCS project also seems to be adopting a hybrid approach. It seeks to combine extended relational algebra or RBU operations with committed choice resolution to reconcile deductive capabilities with systems programming capabilities in a parallel processing framework. Input resolution is possible in terms of RBU operations, and an algorithm is given in [Yokota & Itch 1986] for realising breadth first search by RBU operations. This approach executes RBU operations from within FGHC instead of using Prolog to realise resolution. A major advantage seems to be that it allows a bottom up query evaluation approach to be used, which ensures completeness and termination for query processing. Without restrictions on recursion a top down query evaluation procedure like Prolog’s SLD resolution strategy is not guaranteed to terminate on finite problems and the query strategy is correspondingly incomplete. Top down query evaluation strategies for definite databases like QSQR/SLD [Vieille 1987] which terminate for finite problems exist, but they do not implement efficiently on top of Prolog implementations. However, bottom up query evaluation procedures have their own major disadvantage of not allowing the query to restrict the search space in the way in which top down query evaluation procedures can. Furthermore non-terminating top down query evaluations are amenable to being handled. Chapter six showed how escape exception handling can allow
Prolog to recover gracefully from non-terminating computations which cause stack overflow. This method can serve as an efficient loop detector, and provides a significant programming tool for recognising the non-terminating queries which have to be handled with special strategies.

Calls to \texttt{rbu/1} would serve much the same role in an FGHC program on a PIM as calls to the transiently communicating interface \texttt{prolog/1} would from the CLP engine to a Prolog in a coupled resolution architecture. They both enable a committed choice computation to get back multiple solutions to a query over a clause database as a stream. Furthermore their macroscopic processing structure would also seem to be similar. Each time an \texttt{rbu/1} predicate is executed, a new RBU process is created which is able independently to retrieve and update relations [Yokota et al 1988] p.12. Equally each time a call is made to \texttt{prolog/1} from within a CLP computation on a coupled resolution architecture, a new (lightweight) process is created with its own retrieval and update functionality. However, by going for a dedicated knowledge base approach, the FGCS project is losing a lot of flexibility in programming querying over the knowledge base. The functionality of \texttt{rbu/1} is hard-wired, whereas the functionality of a Prolog computation invoked from a CLP computation is open to a rich and powerful panoply of Prolog computation techniques which can fully exploit its backtracking mechanisms and full logical unification. Furthermore by eschewing an interface to a programmable SLD resolution engine like Prolog, the FGCS project’s approach is losing the rich possibilities of plugging into or-parallel Prolog implementation technology [Warren 1988].

A major advantage to the FGCS project of opting for having a dedicated knowledge base exercised by an \texttt{rbu/1} interface is that the dedicated knowledge base can be optimised for performance at both the software and the hardware level. A major disadvantage is lack of flexibility in programming use of the dedicated knowledge base. This may not be wise in a next generation machine. Lack of flexibility in handling the knowledge base through a high level \texttt{rbu/1} interface may soon be exposed by innovation in query evaluation procedures, parallel algorithms, knowledge representation techniques, and application requirements. Adopting a relational knowledge base approach is also compromising on one of the original themes of the FGCS project of developing a fifth generation machine based upon using logic programming as the underlying computational paradigm.
Appendix A

Language Syntax

**GHC**

\[\text{ghc\_relation} = \text{ghc\_clauses}, '. '\]

\[\text{ghc\_clauses} = \text{ghc\_clause}, [ '.', \text{ghc\_clauses} ]\]

\[\text{ghc\_clause} = \text{literal}, ':-', \text{guarded\_body} |\]

\[\text{guarded\_body} = \text{literals}, '|', \text{literals} | \text{quoted\_vertical\_bar}, '(' \text{literals}, ',', \text{literals}, ')'\]

\[\text{literals} = \text{literal}, [ ',', \text{literals} ] | \text{single\_quote}, ',', \text{single\_quote}, '(' \text{literals}, ',', \text{literals}, ')'\]

**Lingua Franca**

\[\text{lingua\_relation} = \text{lingua\_clauses}, '. '\]

\[\text{lingua\_clauses} = \text{lingua\_clause}, [ ( ':' | '.' ), \text{lingua\_clauses} ]\]

\[\text{lingua\_clause} = \text{lingua\_head}, [ ':-', \text{lingua\_body} ] | \text{quoted\_rule\_symbol}, '(' \text{lingua\_head}, ',', \text{lingua\_body}, ')'\]
lingua_head = atom, ['(', variable_conjunction, ')'] |
prefix_operator, variable |
variable, postfix_operator |
variable, infix_operator, variable

lingua_body = literals | guarded_body

lingua_goals = literals, '.'

Parlog

parlog_relation = mode_declaration, parlog_clauses, '.'

parlog_clauses = parlog_clause, [ ( '.' | ';' ), parlog_clauses ]

parlog_clause = literal, [ ':::', parlog_body ] |
quoted_rule_symbol, ('(', literal, ',', parlog_body, ')')

parlog_body = parlog_literals, [ '||', parlog_literals ] |
quoted_vertical_bar, ('(', parlog_literals, ',', parlog_literals, ')')

parlog_literals = literal, [ ( ',' | '&' ), literals ] |
quoted_and_symbol, ('(', literal, ',', literals, ')')

mode_declaration = 'mode', atom, ['(', modes, ')'] '://' |
'mode', '(' atom, ['(', modes, ')'] ')' '://'

modes = ( '?' | '™' ), [ '://', modes ]
Symbol Syntax

```
literal = atom | structure

term = variable | number | atom | nil_list | string | list | structure | '(', terms, ')

terms = term, [ '(', term ]

list = '[' list_expression, ']

list_expression = term, [ '(', list_expression ] |
                 term, [ '(', term ] |
                 quoted_vertical_bar, '(' term, '(', term, ')

structure = atom, '(', terms, ')') | prefix_operator, term |
            term, postfix_operator | term, infix_operator, term |
            '{', terms, '}

postfix_operator = unquoted_atom

prefix_operator = unquoted_atom

infix_operator = unquoted_atom

variable_conjunction = variable, [ '(', variable_conjunction ]

quoted_and_symbol = single_quote, ( '(', ' | &', single_quote

quoted_rule_symbol = single_quote, ':->', single_quote
```
Lexical Syntax

Tokenising

\[
\text{tokens} = [\text{identifier} | \text{variable} | \text{number}], [\text{after_identifier}]
\]

\[
\text{after_identifier} = (\text{graphic_symbol} | \text{'}), [\text{after_graphic}] |
\]

\[
(\text{quoted_symbol} | \text{string} | \text{solo_symbol} | \text{comment} |
\text{end_of_line_comment} | \text{layout_character}), [\text{tokens}]
\]

\[
\text{after_graphic} = (\text{identifier} | \text{variable} | \text{number}), [\text{after_identifier}] |
\]

\[
(\text{quoted_symbol} | \text{string} | \text{solo_symbol} | \text{comment} |
\text{end_of_line_comment} | \text{layout_character}), [\text{tokens}]
\]

\[
\text{solo_symbol} = (\text{'} | \text}') | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} | \text{'} |
\]

\[
\text{layout_character} = \text{space} | \text{newline}
\]

\[
\text{space} = \text{'} | \text{tab}
\]

Identifiers

\[
\text{atom} = \text{identifier} | \text{quoted_symbol} | \text{graphic_symbol} | \text{'} | \text{'}
\]

\[
\text{unquoted_atom} = \text{identifier} | \text{graphic_symbol} | \text{'} | \text{'}
\]
variable = (big_letter | '_'), {identifier_character}

nil_list = ['[', ']']

identifier = small_letter, {identifier_character}

identifier_character = letter | digit | '_'

graphic_symbol = graphic_character, {graphic_character | '.'} | '.', graphic_symbol

letter = small_letter | big_letter

small_letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

big_letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

graphic_character = '*' | '+' | '-' | '/' | ':' | '<' | '=' | '>' | '?' | '\' | '|' | '˜' | ':' | '@' | '#' | '

Numbers

number = unsigned_number | '‐', number | '‐', ('(', number, ')') |

unsigned_number = digits, ['.', digits], [exponent]

exponent = (E | 'e'), ['+', '-'], digits
digits = digit, { digit }

digit = octal_digit | '8' | '9'

octal_digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'

Quotes

quoted symbol = single_quote, { quoted_character }, single_quote

string = double_quote, { quoted_character }, double_quote

quoted_character = character - ( single_quote | double_quote | '\' | newline ) |
\n, escape sequence

escape sequence = 'b' | 'f' | 'n' | 'r' | 't' | single_quote | double_quote | newline |
\n, octal_digit, octal_digit, octal_digit

single_quote = '

double_quote = "

Comments

comment = '/\n, '*, comment_characters | line_end_comment

line_end_comment = '%', { character }, newline

comment_characters = character - '*, comment_characters |
'*', ( '/' | character - '/', comment_characters )

character = writing_character | newline | space | single_quote

writing_character = '!' | ',' | '#' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '[' | ']' | 'ˆ' | '_' | ''' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '{' | '}' | '˜'
Appendix B

Kelpie Reference Manual

Kelpie is a concurrent logic programming system written in C for single processor Unix systems. It supports the execution of GHC and Parlog, and enables multiple Prolog processes to be invoked concurrently for knowledge based applications. This appendix provides a comprehensive description of Kelpie’s features.

Keywords: and-parallelism, concurrency, logic programming, GHC, Parlog, Prolog

Introduction

Kelpie emulates two related concurrent logic programming languages, GHC and Parlog, and enables concurrent invocation of multiple Prolog processes for knowledge based applications. It is written in C to run under Unix on single processor computers. Currently it runs upon Sun workstations. Kelpie is modelled upon Edinburgh Prolog systems. The interface style, the system predicates available and each system predicate’s action on being satisfied or failing are in most cases very similar. A major difference is that besides succeeding or failing, most system predicates can also suspend waiting for one or more of their arguments to become further instantiated. The syntax and grammar accepted by Kelpie conforms closely to the draft syntax standard for Prolog. Kelpie has been designed to feel familiar to Prolog users. This reference manual does not explain how to program in GHC or Parlog or how to make effective use of Kelpie. A separate work, the Kelpie Users’ Guide, has been written to assist users to use Kelpie to program in GHC and Parlog and to use Kelpie’s main extra feature, its ability to invoke Prolog processes for knowledge base applications.

The Kelpie System

Kelpie is an interactive logic programming system designed to handle execution of four languages.

1) GHC
2) Lingua Franca
3) Parlog
4) Prolog

Kelpie consists of an interpreter of a concurrent logic programming which is a lingua franca of GHC and Parlog and an interface that enables the concurrent execution of multiple Prolog processes from within a lingua franca computation. The lingua franca inference engine is invoked at the top level or by the meta-call
call/3. Prolog executions are invoked by executing the primitive prolog/1 within a lingua franca computation. These Prolog processes are full versions of Prolog-X that can execute on their own or can invoke the facilities of a Prolog Database Machine concurrently. The PDBM Prolog System user manual explains these capabilities [Massey 1989].

GHC and Parlog are translated into the lingua franca on being consulted as GHC or as Parlog, but they are not intertranslatable via their translation into the lingua franca. GHC and Parlog clauses can be listed using various versions of the listing primitive in either their original language or in the lingua franca. Lingua franca clauses can also be consulted directly. The only concurrent logic programming language emulated is the lingua franca. Executing the lingua franca version of a GHC and Parlog clause suffices to emulate the original language clause. Invocation of the system at the top level is in the lingua franca.

The Kelpie system is invoked by the command "kelpie". The following options are available in the command line. Each is a single letter preceded by a minus. The letters may be lower or upper case. They must be followed immediately by an unsigned integer. The integer specifies in thousands of units the storage to be allocated for the relevant storage area. No shorthand way of specifying parameters is supported.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Bytes/Unit</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>20</td>
<td>4</td>
<td>atom table size</td>
</tr>
<tr>
<td>-c</td>
<td>30</td>
<td>4</td>
<td>code area size for the lingua franca</td>
</tr>
<tr>
<td>-g</td>
<td>200</td>
<td>4</td>
<td>global heap size</td>
</tr>
<tr>
<td>-l</td>
<td>50</td>
<td>4</td>
<td>local heap size</td>
</tr>
<tr>
<td>-n</td>
<td>10</td>
<td>4</td>
<td>number heap size for reals</td>
</tr>
<tr>
<td>-p</td>
<td>100</td>
<td>4</td>
<td>process heap size</td>
</tr>
<tr>
<td>-r</td>
<td>5</td>
<td>28</td>
<td>relations table size for the lingua franca</td>
</tr>
<tr>
<td>-s</td>
<td>5</td>
<td>12</td>
<td>suspension table size</td>
</tr>
<tr>
<td>-w</td>
<td>12</td>
<td>4</td>
<td>width of data records</td>
</tr>
<tr>
<td>-x</td>
<td>12</td>
<td></td>
<td>maximum number of simultaneous Prolog-Xs</td>
</tr>
</tbody>
</table>

Thus the command

```
kelpie -g250 -p200 -s10
```

allocates a global heap of 250 000 units, a process heap of 200 000 units and a suspension table of 10 000 units.

The global heap is used to store structures and lists during execution as well as certain non-temporary variables. The local heap is used for various temporary storage purposes including parsed data structures, unification trails, translations of code and all solutions evaluation states. The real number heap is used to store double precision real numbers. A new entry is claimed upon it each time an arithmetic evaluation is/2 or :=/2 binds a left hand side argument with a floating point number. The process heap stores process descriptors and their attached data records created during evaluation. The suspension table records details on
processes suspended upon variables. The amount of units allocated to each can be elicited by the system call \texttt{statistics/0}. If insufficient main memory can be obtained from the UNIX operating system for starting up, start up fails, a warning message is displayed, the call \texttt{statistics/0} is satisfied and Kelpie terminates.

While executing Kelpie uses fixed size units of memory called \textit{data records} to record temporary values needed during computation as well as using the global heap to record more persistent values. Data records are recycled when dispensed with, whereas values persist on the global heap until garbage collection is performed. For this reason it is better for temporary values to be stored in data records where possible to avoid eating up the global heap. Data records are used for various purposes including recording the arguments of goals. The arity of a executable goal must be less than or equal to the current data record size at the time the emulator creates a process descriptor for that goal. Otherwise the emulator cannot continue executing. In addition to recording calling arguments, the data record for a goal is also used to record variables shared between body literals in a clause that do not appear in its head. Unused slots in a goal’s argument record are used for this purpose or if none are left these variables are recorded on the global heap. Thus the size of data records should be set at a value above the highest arity of a goal used in an executing program to avoid excessive use of the global heap. This size should not be much more than is necessary as it will result in a lot of temporary space being temporarily claimed but not used. By using the backtrace to report on how many data records have been allocated during execution and using \texttt{statistics/0} to report on the global heap usage, a user can experiment with data record sizes to find an optimum size for economical memory usage during an application run.

8.6. The Lingua Franca

The lingua franca has the following language characteristics.

1) and-parallel literal conjunction only
2) or-sequential and or-parallel clause search
3) unsafe clause guards
4) input matching performed on all head arguments
5) guard evaluation starts only when input matching succeeds
6) satisfaction of a clause body begins only when the clause is trusted
7) various unification and meta-call primitives

The lingua franca is executed by and-parallel input resolution. In the absence of sequential search restrictions a lingua franca goal is solved by being head argument matched with all lingua franca clauses for that relation. This means unifying with the goal a fresh copy of the clause head of each lingua franca clause so long as this does not bind or share goal argument variables. Other clauses, that might satisfy the head argument matching requirements if the goal was instantiated further, are suspended upon relevant variables and the attempt made to match the clause head with the goal, if and when the variables upon which they are
suspended are instantiated or shared. The guard goals for clauses that pass the head argument matching test are then solved in parallel. Whichever clause’s head argument matching requirements and guard goals for that relation are satisfied first, terminates parallel clause searches and results in the goal being reduced to the goals in the body of the clause. If a guard goal fails in each clause, the goal fails. Where there are sequential search restrictions on clauses, then a lingua franca goal is only matched and unified with the clause heads of all clauses up to the next unencountered sequential search clause terminator. The guards of these clauses are solved in parallel. Only in the event of a guard goal in each such clause failing, are the guards of clauses up to the next unencountered clause search operator examined by head argument matching and guard goal solution.

The evaluation of the guards of lingua franca clauses should not be such as to allow a calling argument variable to be bound or calling argument variables to be shared before the clause is selected for commitment. This restriction is not enforced by the implementation but is left to the programmer to observe. Failure to observe this restriction on guard evaluation for Parlog clauses translated to the lingua franca or for pure lingua franca may result in the clauses not being validly executed. GHC clauses translated to the lingua franca cannot fail to observe this restriction.

The Top Level Interface

The top level interface for Kelpie is written in C. It consists of a simple dialogue cycle. A prompt "| ?- " is issued and user input is parsed by the Kelpie parser. Input must be terminated by a full stop. A following carriage return also has to be given to prod the Unix operating system into passing the input to Kelpie. Input that does not parse correctly causes the display of an appropriate error message, the response no to be given and the initiation of a new query cycle. Correctly parsing input causes the Kelpie system to attempt to create a process for each top level conjunct. Creation of processes can fail for various reasons - because there is insufficient space to build them in or because a literal contains too many variables or is of too high an arity. Evaluation of these processes as a parallel conjunction of literal calls is then attempted. There are four normal outcomes of evaluation.

1) Success yes is reported with top level variable values
2) Failure no is reported
3) Deadlock no is reported with an "All processes suspended" message
4) Error no is reported with an error message

In addition the Kelpie system may go to sleep waiting indefinitely upon input. If no clauses have been defined for a relation, a literal call fails but no warning is given. Only if the trace facility is switched on will a warning be given.
To exit from the top level interface, type the *end of file* character Control Z or use the predicate `halt/0`.

```
?- halt.

%  Kelpie execution halted
```

If the user gets into trouble, a single Control C will halt execution at the beginning of the next emulator cycle. Alternatively to force the emulator to exit back to the top level without regard to preserving the integrity of data areas (including the code area), two Control Cs together should get the user out of most situations. In dire emergencies the quit signal (Control \ on Sun 3s) will ensure the user escapes from any problem the Kelpie system is responsible for back to the Unix operating system.

**Language Primitives**

The lingua franca and Parlog share primitives and use the most extensive set of primitives. GHC uses a restricted set of these available lingua franca primitives. Not all DEC-10 primitives are supported and there are a few extras. The recognised GHC primitives are:
**GHC Primitives**

Exp > Exp1
Exp >= Exp1
Exp < Exp1
Exp =< Exp1
Exp :=: Exp1
Exp =:= Exp1
Term = Term1
Term =\= Term1
Term =.. List
Number := Exp
\+ Goals
Number is Exp
arg(N, Structure, Term)
atom(Term)
atomic(Term)
clauses( Goal, List, Language)
consult(File)
consult(File, Language)
functor(Struct, Functor, Arity)
instream(List)
integer(Term)
Number is Exp
length(List, Integer)
lst(Term)
melt(Term, Term1)
name(Atom, List)
nonvar(Term)
number(Term)
ostream(List)
reconsult(File)
reconsult(File, Language)
system(Atom)
true

Lingua franca system predicates not in this list may not occur in GHC clauses. The emulator will not add clauses containing lingua franca primitives not on this list to its guarded clause database.

**Suspending Primitives**

Some lingua franca primitives can suspend waiting upon their inputs to be further instantiated and several system predicates are multi-moded in an Edinburgh Prolog fashion. There are five types of modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>unify argument</td>
</tr>
<tr>
<td>input</td>
<td>+ instantiation wait input argument</td>
</tr>
<tr>
<td></td>
<td># time of call input argument</td>
</tr>
<tr>
<td></td>
<td>@ conditional input argument</td>
</tr>
</tbody>
</table>
Output arguments are of two kinds. They either unify with the given calling argument or suspend execution until a match condition has been satisfied and then unify with the argument. Input arguments are of three kinds. *Instantiation wait* input arguments suspend execution until the given argument is sufficiently instantiated. *Time of call* input arguments accept the value at the time of call. *Conditional input* arguments cause suspension until some condition is satisfied concerning the argument’s value. These conditions may concern dependencies among arguments to the same relation. Thus conditional input arguments may accept a variable argument as input. Evaluation consists of trying to fit the call to a mode pattern. The first pattern that is matched determines the way the system predicate is evaluated.

The modes of the primitives not available to the user are as follows.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Mode(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term &lt;= Term1</td>
<td>- &lt;= @</td>
</tr>
<tr>
<td>satisfy(Goal, Term)</td>
<td>satisfy(-, -)</td>
</tr>
<tr>
<td>wait(Goal, List, Term)</td>
<td>wait(-, +, -)</td>
</tr>
<tr>
<td>ward(Term, Term1, Term2)</td>
<td>ward(-, +, @)</td>
</tr>
</tbody>
</table>

User available system predicates of arity greater than zero have the following modes
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Mode(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp &gt; Exp1</td>
<td>+ &gt; +</td>
</tr>
<tr>
<td>Exp &gt;= Exp1</td>
<td>+ &gt;= +</td>
</tr>
<tr>
<td>Exp &lt; Exp1</td>
<td>+ &lt; +</td>
</tr>
<tr>
<td>Exp =&lt; Exp1</td>
<td>+ =&lt; +</td>
</tr>
<tr>
<td>Exp =:= Exp1</td>
<td>+ =:= +</td>
</tr>
<tr>
<td>Term == Term1</td>
<td>@ == @</td>
</tr>
<tr>
<td>Term = Term1</td>
<td>@ = @</td>
</tr>
<tr>
<td>Term . List</td>
<td>+ =. -</td>
</tr>
<tr>
<td>Number := Exp</td>
<td>- := +</td>
</tr>
<tr>
<td>Term =&gt; Term1</td>
<td>@ =&gt; -</td>
</tr>
<tr>
<td>+ Goals</td>
<td>+ +</td>
</tr>
<tr>
<td>abolish(Atom, Arity)</td>
<td>abolish(+, +)</td>
</tr>
<tr>
<td>abolish(Atom, Arity, Language)</td>
<td>abolish(+, +, +)</td>
</tr>
<tr>
<td>arg(N, Structure, Term)</td>
<td>arg(+, +, -)</td>
</tr>
<tr>
<td>assert(Clause)</td>
<td>assert(+)</td>
</tr>
<tr>
<td>assert(Clause, Language)</td>
<td>assert(+, +)</td>
</tr>
<tr>
<td>atomic(Term)</td>
<td>atomic(+)</td>
</tr>
<tr>
<td>call(Goals, Result)</td>
<td>call(+, -)</td>
</tr>
<tr>
<td>call(Goals, C, S)</td>
<td>call(+, @, -)</td>
</tr>
<tr>
<td>call(G, C, S, P, Q)</td>
<td>call(+, @, -, +, +)</td>
</tr>
<tr>
<td>clause(Head, Body)</td>
<td>clause(*, -)</td>
</tr>
<tr>
<td>clause(Head, Body, Language)</td>
<td>clause(*, *, -)</td>
</tr>
<tr>
<td>clauses(Goal, List)</td>
<td>clauses(+, -)</td>
</tr>
<tr>
<td>clauses(Goal, List, Language)</td>
<td>clauses(+, -, +)</td>
</tr>
<tr>
<td>close(File)</td>
<td>close(+)</td>
</tr>
<tr>
<td>consult(File)</td>
<td>consult(+)</td>
</tr>
<tr>
<td>consult(File, Language)</td>
<td>consult(+, +)</td>
</tr>
<tr>
<td>display(Term)</td>
<td>display(#)</td>
</tr>
<tr>
<td>display(Term, File)</td>
<td>display(#, +)</td>
</tr>
<tr>
<td>exists(File)</td>
<td>exists(+)</td>
</tr>
<tr>
<td>flush(File)</td>
<td>flush(+)</td>
</tr>
<tr>
<td>freeze(Term, Term1)</td>
<td>freeze(+, -)</td>
</tr>
<tr>
<td>functor(Struct, Functor, Arity)</td>
<td>functor(+, -, +)</td>
</tr>
<tr>
<td>get(Ascii)</td>
<td>get(-)</td>
</tr>
<tr>
<td>get(Ascii, File)</td>
<td>get(-, +)</td>
</tr>
<tr>
<td>get0(Ascii)</td>
<td>get0(-)</td>
</tr>
<tr>
<td>get0(Ascii, File)</td>
<td>get0(-, +)</td>
</tr>
<tr>
<td>help(Atom)</td>
<td>help(+)</td>
</tr>
<tr>
<td>instream(List)</td>
<td>instream(+)</td>
</tr>
<tr>
<td>integer(Term)</td>
<td>integer(+)</td>
</tr>
<tr>
<td>Number is Exp</td>
<td>- is +</td>
</tr>
<tr>
<td>Primitive</td>
<td>Mode(s)</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>length(List, Integer)</td>
<td>length(+, -)</td>
</tr>
<tr>
<td>list(Term)</td>
<td>list(+)</td>
</tr>
<tr>
<td>listing(Language)</td>
<td>listing(+)</td>
</tr>
<tr>
<td>listing(Atom, Language)</td>
<td>listing(+, +)</td>
</tr>
<tr>
<td>melt(Term, Term1)</td>
<td>melt(+, -)</td>
</tr>
<tr>
<td>name(Atom, List)</td>
<td>name(+, -)</td>
</tr>
<tr>
<td>nonvar(Term)</td>
<td>nonvar(+</td>
</tr>
<tr>
<td>number(Term)</td>
<td>number(+</td>
</tr>
<tr>
<td>op(P, Form, Name)</td>
<td>op(+, +, +)</td>
</tr>
<tr>
<td></td>
<td>op(*, *, +)</td>
</tr>
<tr>
<td></td>
<td>op(-, -, +)</td>
</tr>
<tr>
<td>open(File)</td>
<td>open(+)</td>
</tr>
<tr>
<td>open(File, Mode)</td>
<td>open(+, +)</td>
</tr>
<tr>
<td>operators(List)</td>
<td>operators(-)</td>
</tr>
<tr>
<td>outstream(List)</td>
<td>outstream(+)</td>
</tr>
<tr>
<td>primitive(Goal)</td>
<td>primitive(+)</td>
</tr>
<tr>
<td>primitives(List)</td>
<td>primitives(-)</td>
</tr>
<tr>
<td>prolog(Goal)</td>
<td>prolog(*)</td>
</tr>
<tr>
<td>put(Ascii)</td>
<td>put(+)</td>
</tr>
<tr>
<td>put(Ascii, File)</td>
<td>put(+, +)</td>
</tr>
<tr>
<td>read(Term, L, File)</td>
<td>read(-, -, +)</td>
</tr>
<tr>
<td>reconsult(File)</td>
<td>reconsult(+)</td>
</tr>
<tr>
<td>reconsult(File, Language)</td>
<td>reconsult(+, +)</td>
</tr>
<tr>
<td>relation(Atom, Arity)</td>
<td>relation(+, +)</td>
</tr>
<tr>
<td>relation(Atom, Arity, L)</td>
<td>relation(+, +, +)</td>
</tr>
<tr>
<td>relations(List)</td>
<td>relations(-)</td>
</tr>
<tr>
<td>relations(List, Language)</td>
<td>relations(-, +)</td>
</tr>
<tr>
<td>same(Term, Term1)</td>
<td>same(+, +)</td>
</tr>
<tr>
<td>schedule(I, J)</td>
<td>schedule(+, +)</td>
</tr>
<tr>
<td>see(File)</td>
<td>see(+)</td>
</tr>
<tr>
<td>seeing(File)</td>
<td>seeing(-)</td>
</tr>
<tr>
<td>skip(Integer)</td>
<td>skip(+)</td>
</tr>
<tr>
<td>skip(Integer, File)</td>
<td>skip(+, +)</td>
</tr>
<tr>
<td>system(Atom)</td>
<td>system(+)</td>
</tr>
<tr>
<td>tab(Integer)</td>
<td>tab(+)</td>
</tr>
<tr>
<td>tab(Integer, File)</td>
<td>tab(+, +)</td>
</tr>
<tr>
<td>tell(File)</td>
<td>tell(+)</td>
</tr>
<tr>
<td>telling(File)</td>
<td>telling(-)</td>
</tr>
<tr>
<td>var(Term)</td>
<td>var(#)</td>
</tr>
<tr>
<td>withdraw(Head, Body)</td>
<td>withdraw(*, -)</td>
</tr>
<tr>
<td></td>
<td>withdraw(-, *)</td>
</tr>
<tr>
<td>withdraw(Head, Body, L)</td>
<td>withdraw(*, -, +)</td>
</tr>
<tr>
<td></td>
<td>withdraw(-, *, +)</td>
</tr>
<tr>
<td>withdrawall(Language)</td>
<td>withdrawall(+)</td>
</tr>
<tr>
<td>write(Term)</td>
<td>write(+)</td>
</tr>
<tr>
<td>write(Term, File)</td>
<td>write(+, +)</td>
</tr>
<tr>
<td>writeq(Term)</td>
<td>writeq(+)</td>
</tr>
<tr>
<td>writeq(Term, File)</td>
<td>writeq(+, +)</td>
</tr>
</tbody>
</table>

More explicit details of their action are available below.
1. Syntax

A Backus Nauer Form description of acceptable syntax is given in appendix A. The term subset conforms reasonably closely to the Prolog syntax standard. Kelpie recognises seven basic kinds of terms.

1) Atoms character, sign and quoted atoms
2) Variables underline, capital letter and anonymous variables
3) Numbers (signed) integers and real numbers
4) Nil list empty square braces and the null string
5) Lists square brace lists and strings
6) Structures operator and standard form structures
7) Braced terms round or curly braced contexts

These terms are built up into goals and clauses in Kelpie’s four languages.

Atoms

There are three kinds of atoms.

Character Atoms

Character atoms are composed of alphanumerical characters and an underline and start with a lowercase letter. Examples are

apocalypse zodiac1 glyph_of_destiny hex_13_of_the_7

Character atoms can have up to 128 characters in them.

Sign Atoms

Sign atoms have one or more of the following characters in them.

* + - / : < = > ? \ ' " ; @ #

Sign atoms may also have one or more full stop characters in them and may begin with a full stop. However a full stop on its own is not a sign atom. Sign atoms can have up to 128 characters in them. Examples of sign atoms are
Sign atoms do not contain mixes of sign characters and alphanumeric characters.

**Quoted Atoms**

Quoted atoms are single quoted sequences of characters that do not include single or double quotes or backslashes or newline characters. However they may contain well-formed escape sequences. An escape sequence is a backslash \\ followed by a letter. Recognised letters are

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\s</td>
<td>single quote</td>
</tr>
<tr>
<td>\d</td>
<td>double quote</td>
</tr>
<tr>
<td>&lt;nl&gt;</td>
<td>newline</td>
</tr>
</tbody>
</table>

or a backslash \\ followed by three octal digits.

Examples of quoted atoms are

'ancient arts' 'single quote \'' 'rune of ending \032'

Ill-formed escape sequences cause a warning to be issued. Tokenising continues but the attempt to form an escape sequence is ignored.

**Variables**

Named variables begin with a capital letter or an underline. Their body consists of a sequence of alphanumeric characters and underlines. Anonymous variables consist only of an underline. Examples of variables are

_666 _summoner Cabbalistic Cryptogram_of_Illusion

Variables can have up to 128 characters in them. Every occurrence of an anonymous variable indicates a new variable otherwise the same identifier signals the same variable.
Numbers

Numbers are integers or real numbers which may be negatively signed. They may not be positively signed. Examples of integers are

203 01 -34 '-(53) '-(5).

Examples of floating point numbers are

.00203 1.0e+1 -34e2 0.9e-2 '-(53.08)

Floating point numbers use an $e$ to indicate the power of 10 that the preceding value must be multiplied by to give the number’s value.

Kelpie differs from Edinburgh Prolog systems in treating prefix minus as a standard operator. This means that negative numbers are read in as structures of functor name ‘-’. All structures with functor names that are declared to be operators can be read in their operator form as well as in their standard functor form. Thus a negative number can be declared in the rather prolix form of a standard structure. These minus structures are melted on being read in only where their sole argument is an integer or melts to an integer. This means that the following call always fails.

| ?- X = 1, -1 = -X.

It always fails even when the first literal is satisfied first because the second unification is unable to unify a structure with a negative integer atom. Edinburgh Prolog also fails this double goal.

| ?- X = 1, -1 is -X.

By contrast the above unification and arithmetic evaluation succeeds. The range of representable values and further details on numbers are to be found in the section on arithmetic.

Nil list

The nil list is represented by

[]

Unlike Edinburgh Prolog the quoted atom ’[]’ is not the same term as the nil list [].
Lists

Lists are represented by square brace contexts and strings. Square brace lists can either be completely specified or have some elements specified and a tail list given. Examples of square brace lists are

\[
[ \text{Spell} \mid \text{Conjurations} ] \quad [\ 7 \ ] \quad [\ \text{manticore}, \text{ogre} \mid \text{Beasts} ] \quad [\ \text{hope} \mid [\ \text{desire} \mid [\ \text{will} \ ] ] ]
\]

Square brace lists may only have variables, the nil list and lists for their tails on being read in. However execution may bind their tails to any term.

Strings are double quoted sequences of characters. The allowed syntax is the same as for single quoted atoms except that the string is double quoted. They are interpreted as lists of ascii values. Examples of strings and their values

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; &quot;</td>
<td>[32]</td>
</tr>
<tr>
<td>&quot;symbol&quot;</td>
<td>[115, 121, 109, 98, 111, 108]</td>
</tr>
<tr>
<td>&quot;％017n&quot;</td>
<td>[37, 94, 15, 10]</td>
</tr>
</tbody>
</table>

Lists are not converted back into double quoted strings on being output.

Structures

Structures can be represented either in operator form if their principal functor has been successfully declared as an operator, or in standard functor form. In operator form the operator must appear in the position given by its fixity declaration and it must have the right number of arguments in their correct flanking positions. Examples are given later. In standard functor form a structure must have an atom for a functor and a round braced context delimiting its arguments. Examples of standard form structures are

\[
\text{ring(power) } \quad ':-(\text{immortal}, \text{damned}) \quad \text{alchemy(stone + fire/mercury, transform)}
\]

Braces

Curly braced contexts are construed as the structure \{}/1 with the enclosed context as its single argument. Empty curly braces are construed as an atom.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ air, fire, earth }</td>
<td>'{'}(','(air, ',',(fire, earth) ) )</td>
</tr>
<tr>
<td>{}</td>
<td>'{'}</td>
</tr>
</tbody>
</table>
Enclosing any term or complex term in matching pairs of round braces will only ensure it is treated as a single term and will not otherwise affect its interpretation. The empty round braced expression () is not a valid term.

Goals

All syntactically acceptable expressions must be basic terms. Two kinds of terms, atoms or structures, are literal terms when they occur on their own or as principal constituents of a clause. When the principal function of a term with two arguments is ’,’ and its arguments are atoms or structures then the term is a conjunction of literals. Examples of literal terms are

\[
\text{true} \\
\text{enchant(A), conjure(A)} \\
( A = B, B = C, C = D )
\]

A literal term can be a goal or conjunction of goals.

Clauses

Four different kinds of clauses are acceptable to the Kelpie parser. Each must be terminated by a full stop to be read in as a complete term.

GHC clauses

GHC clauses consist of a clause head, a clause guard and a clause body. The :- operator joins the head of the clause to the rest, and the | operator joins the clause guard to the clause body. The clause head must be an atom or a structure and both the clause guard and body must be a literal term. Examples of GHC clauses are

\[
\text{necromancy(Life) :-} \\
\text{death |} \\
\text{sorcery} \\
\text{astrology(sirius) :-} \\
\text{true |} \\
\text{true} \\
\text{occult([A|B], [C|D]) :-} \\
\text{dark_arts(A, C),} \\
\text{A > C |} \\
\text{mystery(C),} \\
\text{occult(B, D)}
\]
Lingua Franca Clauses

Lingua franca clauses consist of a clause head or a clause head and a clause body that is either a literal term or a \( \mid \) separated structure composed of two literal terms. The `:-` operator joins the head of a lingua franca clause to its body if it has such. The clause head must be an atom or a structure. Examples of single lingua franca clauses are

\[
\text{crystal(ball)}\\
\text{projection(Being) :- astral(Being)}\\
\text{closure(Left, Right) :- meld(Left, Right) |}
\quad \text{seal(Left),}
\quad \text{seal(Right)}
\]

An unguarded lingua franca body is construed as meaning that the guard is trivially satisfiable. If no body is given then the guard and body are construed as being trivially satisfiable. Lingua franca clauses form sets of clauses either by being connected by the parallel search terminator \( . \), or the sequential search terminator \( ; \). An example of a set of lingua franca clauses is

\[
\text{spell(Magic, Type) :- summoning(Magic) |}
\quad \text{Type = invocation.}\\
\text{spell(Magic, Type) :- bewitching(Magic) |}
\quad \text{Type = enchantment ;}\\
\text{spell(Magic, Type) :- true |}
\quad \text{Type = diabolic.}
\]

The meaning of the sequential search operator is that the guards of clauses that precede the operator will be searched before the guards of clauses that come after it. Only when the guards of all preceding clauses fail will the guards of clauses after it be searched.

Parlog clauses

Parlog allows literal terms to be conjoined by the sequential conjunction operator \& as well as by the parallel conjunction operator \( . \). Parlog clauses consist of a clause head or a clause head and a clause body that is either a Parlog literal term or a \( \mid \) separated structure composed of two Parlog literal terms. The clause head must be an atom or a structure. Examples of single Parlog clauses are
metamorphosis(philosopher_stone)

transmute(Metal, Sigil) :-
    noble(Metal) &
    chameleon(Sigil)

lore(Arcana, guardian(Adept)) :-
    wise(Adept, Art) |
    knowledge(Arcana, Art),
    power(Art)

An unguarded Parlog body is construed as meaning that the guard is trivially satisfiable. If no body is given then the guard and body are assumed to be trivially satisfiable. Like lingua franca clauses Parlog clauses form sets of clauses either by being connected by the parallel search terminator ; or the sequential search terminator ;. Parlog clauses also must have a mode declaration for the set of clauses. This consists of the mode prefix operator followed by the relation name with a ? or a ^ symbol in every argument place. The whole declaration is terminated by a full stop. An example of a set of Parlog clauses is

```
mode animate(?^).
animate(Substance, necromancy) :-
    living(Substance) |
    true;
    animate(Substance, incarnation).
```

Mode declarations are compulsory.

**Relation to other parsers**

Kelpie’s parser of Prolog-like languages was built to conform to the Prolog syntax standard, although the conformity is not exact. There are several points of conformity with the Prolog syntax standard that are points of departure from the parser of Prolog that appears in C-Prolog, DEC-10 Prolog, Edinburgh Prolog, Quintus Prolog, SICStus Prolog and elsewhere.

1) an operator must be quoted to appear in normal functor form. Quoted atoms are never treated as operators.

2) "." on its own is a clause and goals terminator. Thus typing "true.true." followed by carriage return will produce two top level cycles corresponding to two separate goals. The carriage return is needed in normal interactive mode by the UNIX operating system to jog it into passing the input text to Kelpie. This carriage return will be left in the input stream and so will occur as the first character of the next sequence of characters read in.

3) Except in quoted text, layout characters are nowhere significant even between a functor name and its braced arguments.
4) Only two operators of the same name are allowed and they must not both be infix, or both be prefix or both be postfix. The Kelpie parser performs operator disambiguation rather better than the DEC-10 parser which fails for a surprising number of kinds of cases - particularly with postfix operators.

5) Matching pairs of round braces hide their constituents completely so it is not necessary to put extra braces in `assert(person(X):-rational(X)).`

6) The prefix minus operator is treated exactly like other operators. Therefore minus numbers are structures. These structures are melted to negative integer atoms during a read call.

There are also a number of departures from the Prolog syntax standard. Some of the more significant ones are:

1) System operators are defined like other operators and are revocable unlike in the standard.

2) The range of operator precedences is from 1 to 1500. This allows users to define operators of higher precedences than originally defined operators which only range up to 1250. For example `not/1` might be defined as a prefix operator of precedence greater than 1250.

3) Ambiguity restrictions allow two operators of the same name and the same handedness of associativity.

4) On being read in, lists may only have a variable, the nil list or a list as a tail. Atom tails and structure tails can be only be created by unification.

Other departures from the Prolog standard include using the operators `;`, `:-`, `|` and `&` differently and recognising different kinds of clauses.

**Operators**

Kelpie has a powerful general purpose definite clause programming language parser built into it.

**Predefined Operators**

Kelpie starts up with the following 31 operators declared.
<table>
<thead>
<tr>
<th>Precedence</th>
<th>Fixity</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>xfy</td>
<td>;</td>
</tr>
<tr>
<td>1200</td>
<td>xfx</td>
<td>:-</td>
</tr>
<tr>
<td>1150</td>
<td>fx</td>
<td>mode</td>
</tr>
<tr>
<td>1050</td>
<td>xfy</td>
<td>-&gt;</td>
</tr>
<tr>
<td>1030</td>
<td>xfx</td>
<td></td>
</tr>
<tr>
<td>1030</td>
<td>xfx</td>
<td>:</td>
</tr>
<tr>
<td>1000</td>
<td>xfy</td>
<td>,</td>
</tr>
</tbody>
</table>
| 900        | fy     | \
| 900        | xfy    | &   |
| 800        | xfx    | <=  |
| 800        | xfx    | =>  |
| 700        | xfx    | =   |
| 700        | xfx    | =.. |
| 700        | xfx    | ==  |
| 700        | xfx    | \==|
| 700        | xfx    | \=  |
| 700        | xfx    | :=  |
| 700        | xfx    | :=  |
| 700        | xfx    | is  |
| 700        | xfx    | <   |
| 700        | xfx    | >   |
| 700        | xfx    | =<  |
| 700        | xfx    | >=  |
| 700        | xfx    | =\= |
| 500        | yfx    | +   |
| 500        | yfx    | -   |
| 400        | yfx    | *   |
| 400        | yfx    | /   |
| 400        | yfx    | //  |
| 300        | xfx    | mod |
| 100        | fy     | -   |

Any and all of these operator declarations may be changed. Functor forms can be used instead of declared operators without problem with the sole exception of ",". If the infix declaration for "," is revoked then it will not be possible to read in as input multiple arity structures or multiple element lists. Furthermore Kelpie will use infix commas in writing out multiple arity structures whether an infix comma is currently an operator or not and prefix mode operators in writing out Parlog clauses whether mode/1 is currently a prefix operator or not.

**Parsing operators**

Parsing of tokenised input is performed by taking the parse context and finding the highest precedence operator in it. The highest precedence operator has widest scope.
In this example the precedence of / is 400 and the precedence of + is 500. Thus + is treated as the operator of widest scope. The context is parsed as

'+'( 3, '/'(4, 5) )

Tokens inside matching braces are treated as hidden from the point of view of the encompassing parse context and the braced sequence is treated as an indivisible unit at that level of the parse. Thus

( 3 + 4 ) / 5

is parsed as

'/'( '+'( 3, 4 ), 5 )

Matching pairs of braces - ( ), [ ], { } - cause their constituents to be treated as a single term with the sole exception of the situation where an atom precedes a matching pair of round braces. The context inside is parsed but dominating infix commas are construed as the arguments of a structure so long as the context itself is not braced once more.

Where operators of the same precedence are found together, their fixities are taken into account in determining respective scopes. The fixity declaration uses an f to signify the operator and an x or y to signify the constraints on precedences of operators on respective sides of itself. An x signifies that only lesser precedence operators may occur on that side whereas a y signifies that operators of the same precedence and below may occur on that side. Thus the operators + and - in

3 + 4 - 5

have the same precedence and both have the same fixity yfx. This means that the operator with widest scope must be the - because it can have a same precedence operator on its left hand side while + may not have a same precedence operator on its right hand side. Thus the context is parsed as

'-'('+'(3, 4), 5)

Two operators may share the same name yet have different operator declarations. The operator - is declared as both infix of precedence 500 and prefix of precedence 100. This creates ambiguities that the parser has to resolve. For example in
the parser has to decide that the first occurrence of - is an infix occurrence while the second is a prefix occurrence. Operator disambiguation can get quite complex where there are several ambiguous operators relating to each other. The Kelpie parser manages most multiple operator disambiguations, but there are a few obscure cases it fails to resolve properly. One failure is the following parse with the given declarations

```
| ?- op(400,xf,`**`), op(400,fx,`++`), op(300,xf,`**`), op(300,yfx,`++`).
```

yes
```
| ?- icon ** ++ symbol
```

*** Parsing Error *** Operator(s) incorrectly used
icon**++symbol
Error found on line 1

no
```
| ?-
```

In order to have a deterministic parsing scheme, Kelpie’s parser tries to resolve operator ambiguities by fixity information alone. Only certain patterns of operators with various fixities are possible. For example a postfix operator cannot appear immediately before a prefix operator. All legal patterns of fixity juxtapositions are known to Kelpie’s parser and these patterns are used to determine what fixities are not possible. Kelpie’s parser does not make a guess at which way to construe an ambiguous operator, test the hypothesis and try a different guess if the prior guess fails. That would require Kelpie’s parser to abandon a deterministic scheme of parsing. Kelpie commits itself to using the first fixity consistent construal of an ambiguous operator context that it encounters. As it happens, the first fixity consistent construal examined in the example above is the one that construes `++` as prefix and `**` as infix. Thus Kelpie’s parser attempts to construe the example as

```
`**`(icon, `++`(symbol)).
```

However in the attempt to construe it this way the parser rejects the combination as illegal given their associativity in relation to their precedences. However the alternative of construing `++` as infix and `**` as postfix is perfectly legal. Since there is no trial and error search, this alternative is not considered and the parse fails. The correct construal below is missed.

```
`++`(`**`(icon), symbol).
```

Fixity consistency is a powerful filter of wrong ways to construe ambiguous operators but it is not always enough. It is noteworthy that DEC-10 parser fails on the same example. Users need not worry that this will
affect their programming unless they insist upon using esoteric combinations of several ambiguous operators together.

There are two system predicates concerned with operators - \textit{op/3 and operators/1}. Details can be found in the system predicate details section.

\textbf{Reserved Primitives}

There are two kinds of system predicates. System predicates available to the user and system predicates available only to the Kelpie system. System predicates available only to Kelpie are created while translating GHC and Parlog to the lingua franca. They are not made available to the user to prevent their uncontrolled use. Reserved system predicates are

\begin{align*}
\text{Term} \leq \text{Term1} & \quad \text{unify Term with Term1 without instantiating Term1} \\
\text{satisfy(Goals, Status)} & \quad \text{satisfy Goals grounding Status if successful} \\
\text{wait(Goal, C, Status)} & \quad \text{satisfy Goal when C is a ground term grounding Status if successful} \\
\text{ward(Term, Term1, C)} & \quad \text{copy Term1 into Term if C is not bound otherwise one way unify them}
\end{align*}

Direct use of these predicates is not possible and should not be necessary for the user.

\textbf{\textless /=2}

The reserved system predicate \textless /=2 unidirectionally unifies its first argument with its second without instantiating or sharing variables in its second argument. If in order to unify its arguments, it would have to instantiate variables in its second argument, it suspends upon those variables. This primitive functions transiently, passing bindings from its second argument to its first. These bindings are irrevocably made. It is used by Kelpie to ensure that head argument matching of GHC clauses that have non-empty guards, proceeds in parallel with guard evaluation in their lingua franca equivalents.

\textbf{satisfy}

The reserved system predicate \textit{satisfy/2} is a simple meta-call that waits until its first argument has succeeded or failed. If the goals in its first argument fail, it fails. Otherwise it succeeds if its first argument succeeds and then grounds its second argument. Its function is to send a signal by binding the variable in its second argument when and only when the goals in its first argument are satisfied. It requires its first argument to be bound to one or more literals. It is used by Kelpie as part of the implementation of the GHC runtime suspension test.
wait

The reserved system predicate \textit{wait/3} is a meta-call that waits until its second argument is instantiated to a complete list of ground terms and then attempts to satisfy its first argument. If the attempt to satisfy its first argument fails, it fails. If the attempt succeeds, then \textit{wait/3} grounds its third argument variable and succeeds. The predicate is used by Kelpie to implement sequential conjunctions in Parlog. The control variables in its second argument will be third argument variables of other \textit{wait/3} meta-calls whose first argument goal has to be satisfied before the first argument goal of the \textit{wait/3} in question is satisfied. The Parlog clause

\begin{verbatim}
mode enspell.
enspell :-
  true |
  (invoke, gesture) &
  (conjure, enchant) &
target.
\end{verbatim}

is translated to the lingua franca clause

\begin{verbatim}
enspell :-
  true |
  wait(invoke, [], A),
  wait(gesture, [], B),
  wait(conjure, [A, B], C),
  wait(enchant, [A, B], D),
  wait(target, [A, B, C, D], E).
\end{verbatim}

Kelpie is designed to handle the satisfaction of parallel conjunctions more efficiently than it handles the satisfaction of sequential conjunctions. As the avoidable use of sequential conjunctions gives poorer declarative readings to programs, it is recommended that for both efficiency and semantic reasons, the use of sequential conjunctions should be avoided wherever possible in Parlog programs.

ward

The reserved system predicate \textit{ward/3} performs the run-time suspension test of GHC. Essentially its role is to try to unify its first argument with a consistent copy of its second argument, succeeding only if both arguments are ground and the same. It fails if the unification cannot succeed. So long as it can succeed, it carries on doing this with each change in its second argument until its third argument is a ground term. When and only when this happens, it performs one-way unification between its first two arguments protecting from instantiation but allowing sharing of variables in its second argument. The idea behind it is to protect its second argument from instantiation but to allow consistent copies of bindings to be passed from the second argument to the first.
Thus the GHC clause

\[
\text{exorcise}(A) : -
\text{possessed}(A) \mid
\text{redeemable}(A).
\]

is translated to the lingua franca clause

\[
\text{exorcise}(A) : -
\text{ward}(B, A, C),
\text{satisfy}(\text{possessed}(B), C) \mid
\text{redeemable}(A).
\]

The meta-call \text{satisfy/2} signals to the third argument of \text{ward/3} if and when the user defined predicate \text{possessed/1} succeeds.

<table>
<thead>
<tr>
<th>Calling Values</th>
<th>Output</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ward}(_23, _42, _51)</td>
<td>\text{ward}(_23, _42, _51)</td>
<td>suspend on _42, _51</td>
</tr>
<tr>
<td>\text{ward}(_23, _42, 0)</td>
<td>\text{ward}(_23, _23, 0)</td>
<td>success</td>
</tr>
<tr>
<td>\text{ward}(_23, \text{chimera}, _51)</td>
<td>\text{ward}(\text{chimera}, \text{chimera}, _51)</td>
<td>success</td>
</tr>
<tr>
<td>\text{ward}(\text{chimera}, _42, _51)</td>
<td>\text{ward}(\text{chimera}, _42, _51)</td>
<td>suspend on _42</td>
</tr>
<tr>
<td>\text{ward}(\text{unicorn}, \text{chimera}, _51)</td>
<td>\text{ward}(\text{unicorn}, \text{chimera}, _51)</td>
<td>failure</td>
</tr>
<tr>
<td>\text{ward}(__17, \text{ogre}, [__42, __51])</td>
<td>\text{ward}([\text{imp}, __42], [__51])</td>
<td>suspend on _42</td>
</tr>
<tr>
<td>\text{ward}(_23, [_37, _37], _51)</td>
<td>\text{ward}([_62, _62], [_37, _37], _51)</td>
<td>suspend on _37, _51</td>
</tr>
<tr>
<td>\text{ward}(__a(_23, _23), a(_37, _38), 0)</td>
<td>\text{ward}(__a(_23, _23), a(_37, _38), 0)</td>
<td>suspend on _37, _38</td>
</tr>
<tr>
<td>\text{ward}(1,2, (_26, _26), _31)</td>
<td>\text{ward}(1,2, (_26, _26), _31)</td>
<td>failure</td>
</tr>
</tbody>
</table>

The reserved system predicate \text{ward/3} exhibits the above behaviour.

**Arithmetic and Numerical Comparison Primitives**

Kelpie performs integer and floating point arithmetic. The results of arithmetic expressions are evaluated as integers where possible but are converted into floating point numbers if that is necessary to represent the result of a computation accurately. Floating point numbers are represented by double precision arithmetic. The range of representable integers is

\[-67108863 \leq \text{integer range} \leq +67108863\]
Values outside that range are represented as floating point numbers. The range of representable floating point numbers excluding zero is

\[-1.0 \times 10^{+308} > \text{floating point range} < -1.0 \times 10^{-308} \]

\[1.0 \times 10^{-308} > \text{floating point range} < 1.0 \times 10^{+308} \]

Values outside that range are represented by minus infinity, zero and plus infinity. It is not recommended that huge amounts of floating point arithmetic be done on Kelpie. A new cell on the real number heap is claimed every time the left hand argument of an `is/2` primitive call is instantiated to a real number. Thus only the heap size of instances of floating point numbers can be stored between top level query cycles without overflowing it.

**Arithmetic Evaluation**

Only two synonymous system predicates `is/2` and `=/2` can bind a variable to the result of an arithmetic computation. These predicates evaluate their right hand side and unify the left hand side with the resulting integer or floating point number.

```
?- N is 3 + 4/7.
N = 3.57143
yes
```

All the other arithmetic predicates below evaluate their arguments and succeed or fail depending upon whether the condition is satisfied or not.

**Arithmetic Comparison**

The following arithmetic comparison predicates are available

- `Exp > Exp1`: Exp sums to a larger value than Exp1
- `Exp >= Exp1`: Exp sums to the same/larger value than Exp
- `Exp < Exp1`: Exp sums to a smaller value than Exp1
- `Exp =< Exp1`: Exp sums to the same/smaller value than Exp1
- `Exp =:= Exp1`: Exp sums to Exp1
- `Exp =\= Exp1`: Exp sums not equal to Exp1

All these primitives suspend if either of the arithmetic expressions in their arguments is insufficiently instantiated to be fully evaluated. Variables in arithmetic expressions may be instantiated to integers, to floating point numbers or to arithmetic expressions. Thus the following call succeeds.
A is B + C, C = (4*D)/B, B = 14, D = 7.

A = 16
B = 14
C = (4*7)/14
D = 7

yes

Arithmetic expressions may only contain the following evaluable functions.

- X negation
X + Y addition
X - Y subtraction
X * Y multiplication
X / Y division
M // N integer division
M mod N modulo function
X ^ Y power function
exp(X) exponential
log(X) natural logarithm
log10(X) base 10 logarithm
sqrt(X) square root
sin(X) sine
cos(X) cosine	an(X) tangent
asin(X) arc sine
acos(X) arc cosine
atan(X) arc tangent
floor(X) rounded down value
cputime cpu time (secs)
heapused heap used (bytes)
pi pi
random pseudo-random integer

X and Y are any arithmetic expressions and M and N are arithmetic expressions evaluating to integers. The call

X is cputime.

X = 0.32

yes

gives the cpu time used by Kelpie since start up. This can be used by benchmarking programs. It does not include the cpu time spent by system processes on the Kelpie process’s behalf e.g. paging.
Term Handling Primitives

Kelpie provides the following primitives for assembling and decomposing terms.

- Term = Term1: unify Term and Term1
- Term => Term1: unify Term one-way with Term1 binding only Term1
- Structure =.. List: List is the functor and arguments of Structure
- arg(N, Structure, Term): Term is the Nth argument of Structure
- freeze(Term, Term1): Term1 is a frozen representation of Term
- functor(Struct, Functor, Arity): Struct has Functor and Arity
- melt(Term, Term1): Term1 is a melted representation of Term
- name(Atom, List): Atom is composed of ascii List

They can be used in conjunction with the term matching primitives.

= 

Full unification is performed by Kelpie. Variables bound during the process of unification are reset if the unification fails. Unification never suspends.

=>

The system predicate =>/2 acts exactly like <=/2 except that it one-way unifies in the opposite order. Unlike <=/2, it is available for use by the user. Users should beware that the right hand argument of =>/2 is time of call, and that it is the user’s responsibility to ensure that it is as fully instantiated as intended, when it is called.

=.. 

The system predicate =../2 allows structures to be decomposed into a list of the functor name and its arguments

| ?- coven(celtic, [holly, oak]) =.. List.

List = [ coven, celtic, [holly, oak] ]

yes
| ?- 

or structures to be composed out of a list of an atom and a completed list of terms.
Either the first argument can be used as a
input argument or the second argument can be used as an input
argument. If both are uninstantiated the predicate suspends. Otherwise an instantiated first argument is
expected to be an atom or a structure and an instantiated second argument is expected to be a list. Both
dependencies can be instantiated and then Kelpie checks that the given right hand argument unifies with the
result of decomposing the left hand argument into a list.

arg

The system predicate suspends until its first two arguments are instantiated. It expects an integer N in its
first argument and a structure S in its second argument. Its function is to attempt to unify it third argument
with the $N$th argument of S.

freeze

The system predicate $\text{freeze/2}$ suspends until its first argument is instantiated. On being executed it tries to
unify its second argument with a copy of the first argument except that all variables in the copy are bound
in a consistent fashion to the atoms ... .

The output of $\text{freeze/2}$ can be used as input to $\text{melt/2}$. Together they suffice to copy a term.
functor

The system predicate functor/3 can take as input a structure in its first argument and attempt to unify its second and third arguments with the structure’s functor name and arity

```
?- functor(elixir(magic, invisibility), Name, Arity).
Name = elixir
Arity = 2
```

yes

or it can take as input an atom as second argument and an integer as third argument and attempt to unify its first argument with a structure of that functor name and arity.

```
?- functor(Structure, elixir, 2).
Structure = elixir(_3, _4)
```

yes

The system predicate suspends until either its first argument is instantiated or its last two arguments are instantiated. If all its arguments are instantiated, the predicate verifies that the functor and arity of the structure are correctly given by the last two arguments.

melt

The system predicate melt/2 suspends until its first argument is instantiated. On being executed it tries to unify its second argument with a copy of the first argument except that all atoms in the copy that are bound to the atoms , , , ... are consistently replaced by fresh variables.

```
?- melt(castle("", ""), ("", ""), Term).
Term = castle(_2, _3), (_3, _7))
```

yes

The system predicate melt/2 fails if it encounters a variable as part of the non-variable term in its first argument. The output of freeze/2 can be used as input to melt/2. Together they suffice to copy a term. The output of clauses/2 can also be melted by melt/2.
name

The system predicate name/2 suspends until either of its arguments are instantiated. It expects an instantiated first argument to be an atom and an instantiated second argument to be a list. If its first argument is instantiated to an atom, it tries to unify its second argument with a list of integers that are the ascii values of the characters in the atom’s string value

| ?- name(arcane, List).
List = [97, 114, 99, 97, 110, 101]
yes
| ?- 

or if its first argument is uninstantiated, it tries to unify its first argument with the atom that has the ascii values of the integers in its second argument list.

| ?- name(Atom, [97, 114, 99, 97, 110, 101]).
Atom = arcane
yes
| ?- 

If the first argument is uninstantiated and its second argument is not instantiated to a complete list, the system predicate suspends waiting for its second argument to be instantiated further or for the first argument to be instantiated. It returns failure if an element of the second argument list is not a variable or an integer lying in the range $0 \leq \text{element} < 255$.

**Term Matching Primitives**

The following system predicates are available for term matching.

<table>
<thead>
<tr>
<th>Term == Term1</th>
<th>Term and Term1 are identical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term == Term1</td>
<td>Term and Term1 cannot be identical</td>
</tr>
<tr>
<td>Term = Term1</td>
<td>Term and Term1 cannot unify</td>
</tr>
<tr>
<td>atom(Term)</td>
<td>Term is an atom</td>
</tr>
<tr>
<td>atomic(Term)</td>
<td>Term is an atom, number or []</td>
</tr>
<tr>
<td>integer(Term)</td>
<td>Term is an integer</td>
</tr>
<tr>
<td>length(List, Integer)</td>
<td>List has length Integer</td>
</tr>
<tr>
<td>list(Term)</td>
<td>Term is a list or []</td>
</tr>
<tr>
<td>number(Term)</td>
<td>Term is an integer or a real number</td>
</tr>
<tr>
<td>nonvar(Term)</td>
<td>Term is not a variable</td>
</tr>
<tr>
<td>primitive(Goal)</td>
<td>Goal is a primitive call</td>
</tr>
<tr>
<td>same(Term, Term1)</td>
<td>Term and Term1 match as copies</td>
</tr>
<tr>
<td>var(Term)</td>
<td>Term is a variable at the time of call</td>
</tr>
</tbody>
</table>
None of these predicates tries to instantiate its arguments.

==

The system predicate ==/2 tests whether its arguments are identical.

    | ?- H == I, H = I.
    H = _0.
    I = _0.

  yes
    | ?-.

If they can never be, it fails. If they are identical, it succeeds. Otherwise it suspends.

\==

The system predicate \==/2 tests whether its arguments are identical. If they can never be, it succeeds. If they are identical, it fails. Otherwise it suspends.

    | ?- I \== H.

  *** Execution error *** All processes suspended
  no
    | ?-.

The same job as \==/2 can be done using ==/2 inside negation as failure \+/1. \=/2 is a synonym for \==/2.

\=

The system predicate \=/2 is a synonym for \==/2. Thus it succeeds if its arguments can never unify and fails if they will always unify and suspends otherwise.

    | ?- alike(good, evil) \= alike(A, A).
    A = _3

  yes
    | ?-.

No variables in its arguments are ever bound.
atom

The system predicate atom/1 suspends until its argument is instanitated and then it succeeds if it is a character atom, a sign atom or a quoted atom. It fails otherwise.

?- atom(succubus).
yes
?- .

Empty curly braces {} is an atom also.

atomic

The system predicate atomic/1 suspends until its argument is instanitated and then succeeds if its argument is an atom, an integer, a floating point number or the nil list. It fails otherwise.

?- atomic(-*+*-).
yes
?- .

The empty string "" is atomic because it is read in as [].

integer

The system predicate integer/1 suspends until its argument is instanitated and then succeeds if its argument is an integer atom and fails otherwise.

?- integer(666).
yes
?- .

If the predicate’s argument is instanitated to the structure */1 with an argument, this predicate will fail.

length

The system predicate length/2 suspends until its first argument is instanitated to a complete list or the nil list. Then it attempts to unify the second argument with an integer giving the length of the list.
This predicate expects a variable, list or the nil list as its first argument and a variable or an integer as its second argument.

**list**

The system predicate `list/1` suspends until its argument is instantiated and then succeeds if its argument is a list or the nil list and fails otherwise.

```
| ?- list(L), L = [_|_].
  L = [__1|__2]
  yes
  | ?-
```

A double quoted string is treated as a list of ascii values. Furthermore the quoted atom `'[']` is treated distinctly from the nil list `[]`. The quoted atom `'[']` is not a list and will cause the predicate to fail.

**number**

The system predicate `number/1` suspends until its argument is instantiated and then succeeds if it is an integer or a floating point number and fails otherwise.

```
| ?- number(0.00e-0).
  yes
  | ?-
```

The structure `-/1` unmelted with a number argument is not recognised as a number. Only a `read/1` call or an `is/2` call can extract a number atom from such a structure.

**nonvar**

The system predicate `nonvar/1` suspends until its argument is instantiated and then succeeds.
?- nonvar(L), L = [].
L = []
yes
?- 

This predicate never fails.

**primitive**

The system predicate `primitive/1` suspends until its argument is an atom or a structure and then succeeds or fails depending upon whether its argument is a call to a recognised primitive.

?- primitive(true).

yes
?- 

System reserved primitives like `<=/2` are not recognised.

**same**

The system predicate `same/2` suspends until its two arguments are instantiated and then examines whether its arguments are copies. If they can never be, it fails. If they are copies of each other, it succeeds. Otherwise it suspends upon enough variables in both its arguments to allow it to register when it can succeed or fail.

?- same([A, A], [B, C]), C = B.
A = _0
B = _2
C = _2

yes
?- 

This predicate never binds or shares its arguments. Since when it suspends, it suspends upon a good proportion of the variables in its arguments, it can be computationally expensive to use a lot.

**var**

The system predicate `var/1` takes its argument value at the time of call and succeeds if it is a variable and fails otherwise.
This predicate has time of call semantics and should not be used by programmers who want a good declarative reading for their programs. It is necessary for defining unification and related functions explicitly.

**Consulting and Listing Primitives**

Kelpie supports execution of four logic programming languages. It executes the lingua franca directly, GHC and Parlog indirectly using the lingua franca and Prolog mediately by passing execution to a Prolog system. System calls to Kelpie distinguish between languages using the atoms

<table>
<thead>
<tr>
<th>Atom</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ghc</td>
<td>Guarded Horn Clauses</td>
</tr>
<tr>
<td>kernel</td>
<td>lingua franca</td>
</tr>
<tr>
<td>parlog</td>
<td>Parlog</td>
</tr>
</tbody>
</table>

The following predicates can be used for consulting and listing.

- `consult(File)` consult a lingua franca File
- `consult(File, Language)` consult a Language File
- `listing` list database of lingua franca clauses
- `listing(Language)` list Language database
- `listing(Atom, Language)` list relation name Atom in Language database
- `reconsult(File)` reconsult a lingua franca File
- `reconsult(File, Language)` reconsult a Language File

The default language for consulting and listing is *kernel* and when consulting and listing the lingua franca the last argument may be omitted.

**consult**

There are two system predicates `consult/1` and `consult/2`. Both suspend until their arguments are instantiated. The first argument of each predicate specifies the name of the file to be consulted. The current directory is presupposed unless a full path name is given. If the file name is given as *user* then an interactive consult is initiated. To consult a file named *user* a fuller path name can be given. If the consult predicate has two arguments then the first argument gives the language being consulted. The call
consults a file of Parlog called test into the database. When the language is specified as ghc or parlog, this results in the clauses being translated from their original language into the lingua franca. Kelpie keeps information on the source language of each extant concurrent logic programming language relation. Only one concurrent logic programming language origin is allowed for each relation. Consultation of Prolog clauses into a Prolog system should be done using prolog/1 with the argument being either a list of file names or a consult/1 goal.

**listing**

The three listing system predicates list out the clauses in one of the two databases of clauses. The system predicate listing/0 lists out the whole of the lingua franca database. Variables are systematically replaced by capital letters to improve readability. The output of a listing command redirected to a file is suitable for consulting without further modification.

The system predicate listing/1 suspends until its argument is instantiated. It expects it to be instantiated to a language atom ghc, kernel or parlog. On instantiation, it lists the clauses of the specified language. Where the language is specified to be kernel, the predicate lists all kernel clauses whether they were originally consulted as GHC, Parlog or the lingua franca.

```
| ?- listing(ghc).
exorcise([A|B]) :-
    possessed(A) | exorcise(B).
yes
| ?- listing.
exorcise(C) :-
    [A|B] <= C,
    ward(D, A, E),
    satisfy(possessed(D), E) | exorcise(B).
yes
| ?-
```

The system predicate listing/2 suspends until both its arguments are instantiated. It expects them both to be instantiated to atoms and the second argument to be instantiated to ghc, kernel or parlog. On instantiation it lists the clauses with the first argument relation name in the language of the second argument.
Both listing primitives list GHC and the lingua franca in their full head, guard and body form. Parlog clauses are listed in shorter forms when the guards or the guards and bodies are empty.

In addition to the two consult primitives, there are also two closely related system predicates reconsult/1 and reconsult/2. Both suspend until their arguments are instantiated. The first argument specifies the file to be reconsulted. The file name user initiates an interactive reconsult. If the reconsult predicate has two arguments then the first argument gives the file and the second gives the language being reconsulted. These primitives act just like the consult primitives except that they remove existing clauses (and mode declarations) for a relation before adding new ones.

Database Management Primitives

In addition to the consult and reconsult predicates, several system predicates are available for managing the database of lingua franca clauses.

These predicates correspond closely with analogous Prolog system predicates and enable software development environments and meta-interpreters of GHC, lingua franca and Parlog to execute on top of Kelpie.
abolish

The system predicates `abolish/2` and `abolish/3` suspend until their arguments are instantiated. They expect an atom in their first argument, a small positive integer in their second argument and an atom in their third argument if they have one. On execution the attempt is made to erase all clauses of that atom’s name and that integer’s arity in the relevant database. Where the two argument form is given, the attempt is made to erase that relation in the lingua franca database. The three argument form with the third argument being `kernel` is equivalent to the two argument form.

```prolog
| ?- listing.
cabbalist :-
  true |
  sorceror.
| ?- abolish(cabbalist, 0, kernel).
yes
| ?- listing.
yes
| ?-.
```

Where `ghc` or `parlog` is given in the third argument, the attempt is made to erase a lingua franca database relation of the given name if it originated in the given language. These system predicates succeed only if they erase a relation of that relation name.

assert

The system predicate `assert/1` suspends until its argument is instantiated. It expects its argument to be instantiated to an atom or a structure. On being executed it tries to add the argument as a lingua franca clause to the lingua franca database after any existing clauses for that relation.
The system predicate `assert/2` suspends until its arguments are instantiated. It expects its first argument to be an atom or a structure and its second argument to be an atom. It fails if its second argument is not `ghc`, `kernel`, or `parlog`. The attempt is then made to add the first argument to the appropriate database as a clause of the second argument language. This is done at the end of any existing clauses for that relation. Appropriate translations are made from GHC and Parlog to the lingua franca where necessary. The call will fail if the clause does have a legal form for its language. `assert/2` with the second argument instantiated to `kernel` is equivalent to an `assert/1` with the same first argument.

```prolog
mode transmute(? , ? , _).  
transmute(lead, gold, philosopher_stone).

yes  
|  ?- assert( (transmute(X, gold, midas_touch) :- material(X) | true ), parlog).

X = _0

yes  
|  ?- listing(parlog).

mode transmute(? , ? , _).  
transmute(lead, gold, philosopher_stone).
transmute(A, gold, midas_touch) :-  
    material(A) | true.
```

```prolog
|  ?- listing(parlog).

conjuror :-  
    true | 
    thaumaturge.

|  ?- assert(conjuror :- wizard).

yes  
|  ?- listing.

conjuror :-  
    true | 
    thaumaturge.
conjuror :-  
    true | 
    wizard.

yes  
|  ?-
```
The clause may be a mode declaration if the language is declared to be Parlog. Mode declarations must be given before any clauses for a Parlog relation are declared. Lingua franca and Parlog clause declarations may consist of several clauses so long as each clause has the sequential search operator ; connecting it to the next clause.

**clause**

The system predicate *clause/2* suspends until either its first argument is instantiated or its second argument is instantiated. Instantiated first and second arguments are expected to be atoms or structures. If the predicate is given an instantiated first argument, it unifies its two arguments with the head and body of the first lingua franca clause in the lingua franca database whose head is able to unify with the first argument. If the predicate is only given an instantiated second argument, it unifies its two arguments with the head and clause body of the first lingua franca clause in the database whose body is able to unify with the second argument. *clause/3* acts like *clause/2* except that it expects a language atom *ghc*, *kernel*, or *parlog* in its third argument and uses that to determine the language of the clause head and body unifications.

```
| - listing.
apocalypse :-
   true |
   true.

ectoplasm :-
   true |
   ethereal.
sorcery :-
   energised |
   magic.

yes
| - clause(A, (B | magic)).
A = sorcery
B = energised

| - clause(A, (true | C), kernel).
A = ectoplasm
C = ethereal

yes
| -
```
clauses

The system predicate \texttt{clauses/2} suspends until its first argument is instantiated. It expects it to be an atom or a structure. On getting a literal first argument, the predicate unifies the second argument to a list of all lingua franca clauses for that relation in the lingua franca database. The system predicate unifies its second argument with the nil list if there are no guarded clauses for that relation.

\begin{verbatim}
| ?- clauses(_, _, L).
L = [ (app([_7 | _9], _13, [_7 | _17]) :- (true | app(_9, _13, _17))), app([], _21, _21) :- (true | true))]

yes
| ?-.
\end{verbatim}

The arguments of the first argument goal are irrelevant to determining which clauses are retrieved for that goal. \texttt{clauses/3} acts like \texttt{clauses/2} except that it expects a language atom \texttt{ghc, kernel, or parlog} in its third argument and uses that to determine the language of the clauses retrieved. Where GHC clauses are returned, the variables in each clause are frozen with respect to that clause. They may be melted by using the system predicate \texttt{melt/2}. Only GHC clauses are returned in a frozen form.

relation

The system predicate \texttt{relation/2} suspends until its arguments are instantiated. It expects its first argument to be instantiated to an atom and its second argument to be instantiated to a small positive integer. When its arguments are instantiated, the attempt is made to match the atom and integer arguments with the names and arities of existing lingua franca relations. The predicate succeeds if the relation is found and fails otherwise.

\begin{verbatim}
| ?- relation(diabolist, 3).

yes
| ?-.
\end{verbatim}

\texttt{relation/3} acts like \texttt{relation/2} except that it expects a language atom \texttt{ghc, kernel, or parlog} in its third argument and uses that to determine the language of the relation looked for. The related system predicate \texttt{primitive/1} can be used to recognise primitives.

relations

The system predicate \texttt{relations/1} unifies its argument with a list of the names and arities of currently defined relations in the lingua franca database. A relation may exist with no clauses if it originated as a Parlog relation and still has a mode declaration in force for it.
relations/2 acts like relations/1 except that it expects a language atom ghc, kernel, or parlog in its second argument and uses that to determine the language of the relations retrieved. The related system predicate primitives/1 acts like relations/1 except that it unifies its argument with a list of name/arity structures of all recognised primitives.

withdraw

withdraw/2 and withdraw/3 serve a similar role to retract/1 in Prolog systems. However, as retract/1 retracts by matching its argument only on the heads of clauses and thus does not behave consistently with the argument form of clause/2, a different named system predicate has been provided instead. Database reference numbers for clauses and an erase/1 predicate are not supported by Kelpie for reasons of style.

The system predicate withdraw/2 suspends until either of its arguments are instantiated. Instantiated first and second arguments are expected to be atoms or structures. If the predicate is given an instantiated first argument, it unifies its two arguments with the head and clause body of the first lingua franca clause in the lingua franca database whose head is able to unify with the first argument and withdraws it from the lingua franca database. If the predicate is only given an instantiated second argument, it unifies its two arguments with the head and clause body of the first lingua franca clause in the database whose clause body is able to unify with the second argument and withdraws it from the lingua franca database. withdraw/3 acts like withdraw/2 except that it expects a language atom ghc, kernel, or parlog in its third argument and uses that to determine the language of the clause withdrawn. If there are none, it fails. Otherwise it withdraws the first one and succeeds.
apocalypse :-
  true |
  true.

ectoplasm :-
  true |
  ethereal.

sorcery :-
  true |
  energised,
  magic.

?- withdraw(A, (true | B, C), ghc).
A = sorcery
B = energised
C = magic

yes
?- listing(ghc).

The system predicates withdraw/2 and withdraw/3 fail, if they can find no corresponding clause. Mode declarations cannot be withdrawn because they are characteristics of relations not clauses. However, mode declarations can be abolished with the rest of a relation.

withdrawall

The system predicate withdrawall/0 clears the lingua franca database of all guarded clause relations.

?- withdrawall.

yes
?- listing.

yes
?.
All space previously occupied by guarded clauses is recovered. Withdrawing all clauses during execution has unpredictable effects.

The related system predicate `withdrawall/1` waits until its argument is instantiated. It expects it to be instantiated to a language atom `ghc, kernel, or parlog`. It then clears the relevant relations for that language out of one or other of its two clause databases.

**Input/Output Primitives**

Kelpie allows users to use the following system calls to control input and output.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close(File)</td>
<td>close the File</td>
</tr>
<tr>
<td>display(Term)</td>
<td>write Term immediately in standard form to output stream</td>
</tr>
<tr>
<td>display(Term, File)</td>
<td>write Term immediately in standard form to File</td>
</tr>
<tr>
<td>exists(File)</td>
<td>File exists</td>
</tr>
<tr>
<td>flush(File)</td>
<td>flush output to File</td>
</tr>
<tr>
<td>get(Ascii)</td>
<td>asynchronously input next printing Ascii on input stream</td>
</tr>
<tr>
<td>get(Ascii, File)</td>
<td>asynchronously input next printing Ascii from File</td>
</tr>
<tr>
<td>get0(Ascii)</td>
<td>asynchronously input next Ascii on input stream</td>
</tr>
<tr>
<td>get0(Ascii, File)</td>
<td>asynchronously input next Ascii from File</td>
</tr>
<tr>
<td>nl</td>
<td>write a newline character to output stream</td>
</tr>
<tr>
<td>nl(File)</td>
<td>write a newline character to File</td>
</tr>
<tr>
<td>open(File)</td>
<td>open File for reading and writing</td>
</tr>
<tr>
<td>open(File, Mode)</td>
<td>open File for reading and writing in Mode</td>
</tr>
<tr>
<td>read(Term, L, File)</td>
<td>asynchronously read Term with L variable matches from File</td>
</tr>
<tr>
<td>see(File)</td>
<td>open File as input stream</td>
</tr>
<tr>
<td>seeing(File)</td>
<td>File is input stream</td>
</tr>
<tr>
<td>skip(Ascii)</td>
<td>asynchronously skip input up to Ascii</td>
</tr>
<tr>
<td>skip(Ascii, File)</td>
<td>asynchronously skip input in File up to Ascii</td>
</tr>
<tr>
<td>tab(Integer)</td>
<td>write Integer number of spaces to output stream</td>
</tr>
<tr>
<td>tab(Integer, File)</td>
<td>write Integer number of spaces to File</td>
</tr>
<tr>
<td>tell(File)</td>
<td>open File as output stream</td>
</tr>
<tr>
<td>telling(File)</td>
<td>File is current output stream</td>
</tr>
<tr>
<td>write(Term)</td>
<td>write Term in operator form to output stream</td>
</tr>
<tr>
<td>write(Term, File)</td>
<td>write Term in operator form to File</td>
</tr>
<tr>
<td>writeq(Term)</td>
<td>write Term in operator form with needed quotes to output stream</td>
</tr>
<tr>
<td>writeq(Term, File)</td>
<td>write Term in operator form with needed quotes to File</td>
</tr>
</tbody>
</table>

Kelpie also supports two GHC-origin input/output meta-primitives `instream/1` and `outstream/1` to satisfy input and output goals in order.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instream(List)</td>
<td>satisfy List of input/output goals in order</td>
</tr>
<tr>
<td>outstream(List)</td>
<td>satisfy List of output goals in order</td>
</tr>
</tbody>
</table>

GHC clauses may only use them to handle input and output.
close

The system predicate `close/1` suspends until its argument is instantiated. It expects an atom as its argument. It succeeds if it is able to close the given file and fails otherwise.

```
|  ?- close('/users/karen/tmp').
  yes
|  ?-
```

Kelpie will close all open files on halting or aborting. The prior status of devices will be restored on being closed.

display

The system predicate `display/1` writes its sole argument upon the current output stream in standard form immediately upon being executed.

```
|  ?- display(Term).
  _0
  Term = _0

  yes
|  ?-
```

The system predicate `display/2` suspends until its second argument is instantiated. It expects an atomic file name as its second argument. It then writes out to the file the value of the first argument in standard functor form without regard to operator declarations.

```
|  ?- display(1*2/3//4+5, user).
  '+/1 //'('*'(1, 2), 3), 4), 5)
  yes
|  ?-
```

Standard output is signified by the file "user".

exists

The system predicate `exists/1` suspends until its argument is instantiated. It expects an atom as its argument. It succeeds if a file of that name exists and fails otherwise.

```
|  ?- exists('/users/john/tmp').
  yes
|  ?-
```
The file will be searched for in the current directory unless a fuller path name is given.

**flush**

The system predicate `flush/1` suspends until its argument is instantiated. It expects an atomic file name as its argument. It then attempts to flush output to that file. The file must be open for writing or the call fails.

```prolog
?- flush(user).
yes
?- flush(user).
```

The file will be searched for in the current directory unless a fuller path name is given.

**get**

The system predicate `get/1` attempts to read the next printing character from the current input stream asynchronously. The attempt is made to unify the argument with the character’s ascii value and the call succeeds or fails with that unification. No prompt is given.

```prolog
?- get(Asc).
B
Asc = 66
yes
?- get(Asc).
```

If `get/1` fails, the input character is not put back in the input stream.

The system predicate `get/2` suspends until its second argument is instantiated. It expects its first argument to be uninstantiated or instantiated to an integer and its second argument to be bound to an atomic file name. It then attempts to read the next printing character from the file given by its second argument asynchronously. The file must be open for reading or the call immediately fails. If a character is available then the attempt is made to unify the first argument with the character’s ascii value and the call succeeds or fails with that unification. If no printing character is available then the process suspends until a character becomes available whereupon the goal wakes up and is executed as before.

**get0**

The system predicate `get0/1` attempts to read the next character from the current input stream asynchronously. The attempt is made to unify the argument with the character’s ascii value and the call succeeds or fails with that unification. No prompt is given.
Users of `get0/1` should remember that the `read/1` predicate used by the cycling top level interpreter only reads up to and including the terminating full stop. The subsequent carriage return is needed to prod the operating system into passing the input to Kelpie but it is not part of the term. Consequently the first unread character of the input stream after typing in `get0(A)` followed by carriage return is the carriage return left over from the last query cycle which will be returned immediately as above.

The system predicate `get0/2` suspends until its second argument is instantiated. It expects its first argument to be uninstantiated or instantiated to an integer and its second argument to be bound to an atomic file name. It then attempts to read the next character from the file given by its second argument asynchronously. If `get0/2` fails, the input character is not put back in the input stream.

**instream and outstream**

The system predicates `instream/1` and `outstream/1` suspend until their argument is instantiated. They expect their argument to be a list. If it is the nil list, they succeed. They treat elements of the list as a stream and process them sequentially as input and output goals one at a time. The current element of the stream is the first unprocessed goal. They suspend if the current element of the list is uninstantiated. The elements of the stream are ordinary input and output system calls. `instream/1` takes any input and output goal whereas `outstream/1` only takes output goals.

```prolog
| ?- outstream([ write('beware', user), nl(user), tab(3, user), write('runes', user)]).
  beware
  runes
yes  |
?-
```

These predicates fail if they encounter illegal goals or if a goal fails. They succeed when all the goals in the stream have been satisfied and they encounter the completed end of a pure list. The goals `display/1` and `display/2` which are normally time of call on their first argument behave a little differently when executed by `instream/1` and `outstream/1` as they suspend until their first argument is instantiated.

**nl**

The system predicate `nl/0` outputs a carriage return character on the current output stream. The related system predicate `nl/1` suspends until its argument is instantiated. It expects its argument to be instantiated to an
atomic file name. On execution it outputs a carriage return to the given file which must be open for writing.

**open**

The system predicate `open/1` suspends until its argument is instantiated. It expects an atom as its argument. It succeeds if it is able to open the given file for reading and writing and it fails otherwise.

```
?- open('/dev/tty3').
yes
?-.
```

Unlike `see/1` or `tell/1` the file opened is not made into the current input or output stream. If the file is already open but not for both reading and writing, the call closes the file and reopens it for reading and writing.

The attempt to open a device for reading and writing results in the device being associated with the kelpie process group and any processes that the device was formerly associated with being sent a stop signal. This should prevent external process interference with input. On being closed, the device is restored to its original modes, it is re-associated with its former process group and a continue signal is sent to that process group. When the device is opened, it is put into the line oriented cooked mode, the block and resume output control characters ‘S’ and ‘Q’ are disabled, the file pointer is put to the end of the stream, and the asynchronous input/output flag is set. The input file `user` is already open in the right fashion to the kelpie process and cannot be closed.

The system predicate `open/2` suspends until both its arguments are instantiated. It expects both to be bound to atoms. It succeeds if it is able to open the given file for reading and writing in `cooked` or `cbreak` mode depending upon the second argument being `line` or `character`. It fails otherwise.

```
?- open('/dev/tty3', character).
yes
?-.
```

Character mode means that as each character is typed in, the Kelpie system deals with it directly rather than having to wait to receive the character until a newline or end-of-file character is typed in. This is important for the programming of single key responsive interfaces.

**read**

The system predicate `read/3` suspends until its third argument is instantiated. It expects its third argument to be bound to an atomic file name that is open for reading. On execution it attempts asynchronously to read a term from the given file and tries to unify its first two arguments with the term and a list of associations between user names and internal representations of variables. If the first argument unification fails, no
characters are replaced in the input stream. No prompt is issued.

```
| ?- read(A, L, user).
damned(Soul, Deity) :- evil(Soul), displeased(Soul, Deity).

A = damned(_8, _9) :- evil(_8), displeased(_8, _9).
L = [ 'Soul' = _8, 'Deity' = _9 ]

yes
| ?-
```

This predicate enables Kelpie to support variable reporting facilities in meta-interpreters and to support multiple interfaces at the same time. The system predicate `read/3` reads up to and including the character immediately following a full stop `. `. This is important for recognising the end of a term in a simple fashion. Syntax errors cause the `read/3` call to fail. No error messages are given. The goal remains suspended until it has read a term in from the given file.

**see**

The system predicate `see/1` suspends until its argument is instantiated. The argument is expected to be an atom. Kelpie attempts to make a file of the given atom name into the current input stream. If the attempt fails because the file is not found or too many files have been opened or whatever, the call fails. Devices opened for input are treated as described for the `open/1` predicate.

**seeing**

The system predicate `seeing/1` unifies its argument with the current input stream.

```
| ?- seeing(File).
File = user.

yes
| ?-
```

The default input stream name is `user`.

**skip**

The system predicate `skip/1` suspends until its argument is instantiated. It expects it to be bound to an integer between 0 and 255. On execution Kelpie asynchronously discards input from the current input stream up to and including the next character of its argument’s ascii value.
The system predicate `skip/2` suspends until its arguments are instantiated. It expects its first argument to be bound to an integer between 0 and 255 and its second argument to be bound to an atomic file name. The given file should be open for reading or the goal fails. On execution Kelpie asynchronously discards input from that File up to and including the next character of its first argument’s ascii value.

**tab**

The system predicate `tab/1` suspends until its argument is instantiated. It expects it to be bound to a positive integer. On execution, it outputs that number of space characters to the current output stream.

The system predicate `tab/2` suspends until its arguments are instantiated. It expects its first argument to be a positive integer and its second argument to be an atomic file name. The file should be open for writing or the goal fails. It then outputs that number of space characters to the given file.

**tell**

The system predicate `tell/1` suspends until its argument is instantiated. It expects it to be an atom. Kelpie attempts to make a file of the given atom name into the current output stream. The following calls will output the lingua franca database to a file called "program" in the current directory.

```prolog
| ?- tell(program), listing, tell(user).
yes
| ?-
```

If the attempt fails because the file is not found or too many files have been opened or whatever, the call fails.

**telling**

The system predicate `telling/1` unifies its argument with the current output stream.

**write**

The system predicate `write/1` suspends until its argument is instantiated. Then it writes that term to the current output stream using operator declarations currently in force.
The system predicate \texttt{write/2} suspends until its arguments are instantiated. It expects its second argument to become bound to an atomic file name. That file should be open for writing. Then it writes its first argument to the file using operator declarations currently in force.

\begin{verbatim}
| ?- write('Alchemy is Gramarye of the elements', user).
Alchemy is Gramarye of the elements
yes
| ?-
\end{verbatim}

Variables are written out with reference to their internal forms. An underline number variable like \_31 refers to the 32nd cell on the global heap. A capital V number refers to a data record location in the process data area.

\textbf{writeq}

The system predicate \texttt{writeq/1} suspends until its argument is instantiated. Then it writes that term to the current output stream using operator declarations currently in force quoting atoms that need to be single quoted to be read in as atoms.

The system predicate \texttt{writeq/2} suspends until its arguments are instantiated. It expects its second argument to become bound to an atomic file name. That file should be open for writing. Then it writes out its first argument to the current output stream using operator declarations currently in force.

\begin{verbatim}
| ?- writeq('Alchemy is Gramarye of the elements', user).
'Alchemy is Gramarye of the elements'
yes
| ?-
\end{verbatim}

Atoms that need to be quoted to be read in as atoms are output with enclosing single quotes.

\textbf{Operator Handling Primitives}

Two system predicates are available for handling operators.

\begin{verbatim}
op(P, Form, Name) declare Name as Form operator of precedence P
op/3 of precedence 0 removes an operator declaration
operators(List) unify List with current operator declarations
\end{verbatim}

The earlier section on syntax illuminates differences between Kelpie’s parser, the Prolog syntax standard and the DEC-10 parser.
The system predicate `op/3` can match with, remove or declare an operator. It suspends until its third argument is instantiated. It fails if its third argument is not an atom or must be quoted to be read as an atom. The system predicate expects its second argument either to be uninstantiated or to be instantiated to one of the fixity atoms below.

<table>
<thead>
<tr>
<th>Fixity</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>xfx</td>
<td>infix</td>
<td>non</td>
</tr>
<tr>
<td>xfy</td>
<td>infix</td>
<td>right</td>
</tr>
<tr>
<td>yfx</td>
<td>infix</td>
<td>left</td>
</tr>
<tr>
<td>fx</td>
<td>prefix</td>
<td>non</td>
</tr>
<tr>
<td>fy</td>
<td>prefix</td>
<td>right</td>
</tr>
<tr>
<td>xf</td>
<td>postfix</td>
<td>non</td>
</tr>
<tr>
<td>yf</td>
<td>postfix</td>
<td>left</td>
</tr>
</tbody>
</table>

`op/3` also expects its first argument either to be uninstantiated or to be instantiated to a positive integer between 0 and 1500. When its third argument is an atom, the attempt is made to match it with the name of a currently declared operator. If it is the name of a currently declared operator, the attempt is made to unify its second argument with the fixity of that operator. If two operators of the same name are currently in force, then if the first fails to have a fixity that unifies with the second argument, the attempt is made to unify the fixity of the second operator with the second argument. If one or other succeeds, then the first argument is examined. If it is instantiated to zero, that operator is removed from being in force and the call succeeds.

```
| ?- write('*'(1,2), user).
1*2
yes
| ?- op(0, yfx, '*').
yes
| ?- write('*'(1,2), user).
*(1,2).
yes
| ?- ...
```

Otherwise the attempt is made to unify the first argument with the matching operator’s precedence. If that succeeds, the call succeeds.
| ?- op(Precedence, Fixity, '===').

Precedence = 700
Fixity = xfx

yes
| ?- 

If it fails and a second operator declaration of the same name is in force, then if its fixity and precedence unify with respective arguments, the call succeeds with those unifications.

If an operator has a third argument name that matches with an existing operator but has arguments that fail to unify with the one or other or both of the two preceding arguments, then the preceding arguments are examined. If one or other of the preceding arguments is uninstantiated, the call suspends on that variable. Otherwise and if the name fails to match a currently declared operator name, the attempt is made to declare a new operator of that name.

| ?- write('**(1,2), user).
**(1,2)
yes
| ?- op(500, yfx, **). 

yes
| ?- write('***'(1,2), user).
1***2.
yes
| ?- 

New operator declarations that meet argument range criteria succeed or fail depending upon the rule that only two operators are allowed in force at the same time and must not both be \textit{prefix, infix or postfix}.

\textbf{operators}

The system predicate \texttt{operators/1} unifies its argument with a list of current operator declarations in force. It is provided to allow users to scrutinise operator declarations in force and to enable environments built on top of Kelpie to manage operator declarations.

\textbf{Meta-call Primitives}

Six meta-call predicates are supported by Kelpie.
The last two are reserved system predicates described earlier. They fail if their child or children fail. The first four meta-calls can succeed even though their children fail.

Negation as failure is implemented by \(+/1\). This predicate suspends until its argument is instantiated. It expects its argument to be an atom or a structure. It treats its argument as a goal or a conjunction of goals if the principal functor is ",", and spawns processes to evaluate them. It succeeds if they fail and fails if they succeed.

\(\text{? - } + 1 = 2.\)

\(\text{yes}\)

\(\text{? -}\)

It does not suspend until all the variables in its goals’ argument are instantiated and variables may be bound in its goal argument even though the meta-call fails. Thus it is not a semantically clean version of negation as failure.

**call**

The two argument meta-call \(\text{call}/2\) suspends until its first argument is instantiated. It expects its first argument to be an atom or a structure. It treats its first argument as a goal or a conjunction of goals if the principal functor is ",," and spawns processes to evaluate them. It tries to unify its second argument with \text{yes} if the goal(s) succeed or with \text{no} if one fails.

\(\text{? - call(nonvar(_), B).}\)

\(B = \text{no}\)

\(\text{yes}\)

\(\text{? -}\)

This meta-call does not always succeed. It will fail if its arguments are of the wrong types, or if its second argument does not unify with the result of computing the goals in the first argument.

The system predicate \(\text{call}/3\) implements Parlog’s stream controlled meta-call. It differs in a few respects from SPM’s implementation of the stream controlled meta-call [Gregory et al 1989]. The predicate suspends until its first argument is instantiated to a goal or parallel conjunction of goals. It expects its first argument to be an atom or a structure and its second and third arguments to be variables or lists. The second
argument is the meta-call’s control stream and the third argument is its status reporting stream. The status reporting stream produces the messages

       deadlocked  failed  resumed  stopped  succeeded  suspended

The control stream accepts the messages

       resume  stop  suspend

The messages *deadlocked, failed, stopped or succeeded* are respectively echoed when the goal deadlocks, fails, is stopped or succeeds. The messages *suspended* and *resumed* are echoed when the corresponding control messages *suspend* and *resume* are received. Redundant control messages like several *resume* messages in a row are ignored. Only the first *resume* is reacted to. Invalid argument types and invalid stream elements cause the meta-call to fail.

The meta-call is implemented by recursively calling the C function that executes the process tree. Thus a separate and disjoint process tree is set up for each meta-call. Execution of these process trees continues only for a given quota of reduction steps and then control is returned to the processing level where the meta-call originated and the meta-call is rescheduled to the end of the current scheduling ring for that level. This ensures that other runnable meta-calls have a fair chance of being executed as well. The meta-call never suspends upon its children. The meta-call process suspends upon a variable, suspends upon relevant input streams, or busy waits in the sense of being rescheduled to the end of the scheduling ring of the tree that originated it.

The system predicate *call/5* behaves exactly like the stream controlled meta-call *call/3* except that it suspends until its extra two arguments are instantiated. The fourth argument is expected to be an integer and the meta-scheduling quota for the meta-call is set to that value, and reset to that value if the meta-call is reawakened after being meta-suspended upon a context switch. The fifth argument is expected to be an integer and the scheduling quota for the goals under that meta-call is set to that value. The five argument stream controlled meta-call *call/5* makes programmable load balancing possible.

**Prolog Interface**

Kelpie has the facility to invoke multiple Prologs concurrently. Each separate Prolog is a Unix process communicating with the Kelpie Unix process via pipes. Each lingua franca meta-call can only be associated with one Unix process executing Prolog at a time. Multiple calls can be made to that Unix process within the one lingua franca meta-call but only one is serviced at a time. The view of the Prolog database persists between these calls and may be updated and altered by them. Upon finally exiting from the meta-call, the Unix process executing Prolog is shut down. To run several Prolog invocations at the same time, each
invocation must be made within a separate meta-call. Standard output from each Prolog process goes to the control terminal of the Kelpie process. Standard input to the Prolog process is used for communication with the Kelpie system. Interactive tracing and interactive consultation are not possible with these Prolog systems. Program debugging should be done with a stand alone Prolog system. The actual Prolog system invoked is the enhanced version of Prolog-X that can be used in conjunction with Heriot-Watt university’s Prolog Database machine. Full details on that Prolog system can be found by consulting its manual.

The system predicate `prolog/1` suspends until its argument is instantiated. It expects it to be an atom, a non-empty list or a structure. It then attempts to find a single satisfier of that goal over the Prolog system’s clause database.

```
| ?- prolog([person]).
Consulting test..... done in 114 centi-seconds
yes
| ?- prolog(listing).

person(god).
person(A) :-
    angel(A).
person(A) :-
    rational(A),
    animal(A).

angel(gabriel).
angel(satan).
rational(socrates).
rational(aristotle).

animal(socrates).
animal(aristotle).
animal(bucccephalus).

yes
| ?- prolog(bagof(X, person(X), L)).

X = _0
L = [ aristotle, socrates, satan, gabriel, god ]

yes
| ?-
```

In the example above a Prolog process is created, if it did not exist before, and clauses in the file `test` are consulted into its database. The second call lists the clauses in the Prolog system’s database. The third call performs an exhaustive search over the Prolog system’s database. The Prolog system persists between calls because the top level executes under the same meta-call. Multiple calls to the same Prolog process can be
made by a conjunction of calls

```
| ?- prolog(person(X)), prolog(rational(Y)).
X = god
Y = socrates
yes
| ?- 
```

although only one is executed to completion at a time. To achieve real concurrency, invocations to `prolog/1` must be made under different meta-calls.

```
| ?- call( prolog( [[test],person(god)]) ), _, _
  call( prolog( [[test],person(gabriel)]) ), _, _.
```

As the top level is also under the top level meta-call, this could also be achieved by

```
| ?- prolog( [[test],person(god)])
  call( prolog( [[test],person(gabriel)]) ), _, _.
```

The first example creates and kills two whole Unix processes just to execute two small pieces of Prolog code, whereas the second example creates at least one Unix process and kills at most one Unix process. In order to allow multiple Prolog processes to persist and to be periodically but not continually exercised, their associated meta-call must be kept from terminating. This can be done by suspending the meta-call on further input from an input stream or making it busy wait.

**Miscellaneous Primitives**

**false**

The system predicate `false/0` always fails.

```
| ?- false.
no
| ?-
```

It is called `false` rather than `fail` because it is the antithesis of `true`.

**halt**

The system predicate `halt/0` closes all open files and terminates execution. It is not good programming practice to include a `halt` call in a program. Typing the *end of file* character Ctrl-Z at the interface invokes `halt/0`. If Kelpie receives certain ominous Unix operating signals like a segmentation fault error warning it calls
halts/0.

help

The system predicate help/0 displays details of available primitives. Its function is to supply a fast aide-memoire and not to substitute for this manual.

?- help.

Detailed online help on particular primitives is given by help/1.

?- help(true)

true always succeeds

yes
?- .

The primitive expects its argument to become instantiated to an atom.

primitives

The system predicate primitives/1 unifies its argument with a list of all recognised primitives.

?- primitives(L)

Each primitive is represented in the list by a name/arity structure e.g. true/0.

schedule

The system predicate schedule/2 is provided to enable knowledgeable users to tune the execution characteristics of Kelpie to get better performance out of it relative to their application. Using it users can choose their own values for the meta-call scheduling quota and the scheduling quota. Inexperienced users need not worry about using this primitive as Kelpie uses adequate defaults for their values.

The primitive suspends until its arguments are instantiated. It expects its arguments to be instantiated to positive integers greater than zero. If the values are legal then the meta-call scheduling quota is reset to the first argument’s value and the goal scheduling quota is reset to the second argument’s value.

?- schedule(100,10)

yes
?- .
The current values of these parameters can be inspected by executing the system predicate `statistics/0` and examining what the early entries are in the displayed figure.

Kelpie executes using bounded depth first scheduling. To achieve boundedness each process and each meta-call is given a quota of execution cycles it is allowed before being scheduled to the end of the runnable ring. The amount of these resources left is reduced each time the process is executed until all the resources are used up. At that point the process is given a whole new quota and is scheduled last. Each child of a process inherits the decremented value of its parent’s resources. The effect of giving a lower value to the quota is to change the balance of scheduling towards breadth first search. A quota of 1 achieves pure breadth first search. Meta-calls have a meta-scheduling quota that specifies the number of execution cycles that processes are allowed to execute for in satisfying that meta-call. When the meta-resources are exhausted, the meta-call is rescheduled to the end of the runnable ring for the process tree it was called from and other processes on that tree get executed instead.

Many programs will execute slower if the scheduling quota is made very small. If the scheduling parameters are made very large then scheduling will not be balanced. Processes that are resumed after being suspended upon a child, a variable or an input stream are scheduled for immediate execution with a completely new quota of resources.

**primitives**

The system predicate `primitives/1` unifies its argument with a list of all recognised primitives in the form Name/Arity. It does expect its argument to be bound to an atom or a structure.

**statistics**

The system predicate `statistics/0` displays the current values in units of use of the main data storage areas on the output stream.
| ?- statistics.

Parameter | Value
---|---
Data record size | 12
Metacall Scheduling Quota | 300
Scheduling Quota | 60

| Memory Area | Size | Used |
---|---|---
Code Area | 30000 | 0
Global Heap | 200000 | 0
Relations | 5000 | 0
Local Heap | 50000 | 0
Operator Table | 128 | 30
Process Heap High | 99988 | 0
Process Heap Low | 100000 | 12
Reals Heap | 10000 | 0
String Pockets | free memory | 16
Suspension Table | 5000 | 0
Symbol Table | 20000 | 2

Run time | 0.06 sec.

yes
| ?-

The run time is the elapsed cpu time since start up. Space for strings is allocated dynamically and the total space occupied is indicated by *String Pockets* in bytes.

**system**

The system predicate *system/1* suspends until its argument is instantiated. It expects its argument to be instantiated to an atom. Unlike Edinburgh Prolog systems, it does not accept a double quoted string as argument.

| ?- system(date).

Thu May 5 12:29:56 BST 1989

yes
| ?-

This predicate passes the string value of the atom to the Unix operating system and succeeds if the operating system call succeeds. Otherwise it fails.
true

The system predicate true/0 always succeeds. Certain Kelpie emulator execution optimisations are based upon looking for lone true literals in the guard or body of a clause. Using trivially true system calls other than true like 1=1 as dummy fillers for empty guards or bodies is not recommended.

Tracing Primitives

trace

Kelpie has tracing facilities for following the execution of the lingua franca. The interactive trace system is normally switched off. It may be switched on by executing the call

```
| ?- trace.
  success trace
   yes
| ?-
```

What is displayed is the second and result half of a switched on trace. The interactive trace can be switched off in the same way except that a carriage return must be typed to advance from the line displayed to the result.

```
| ?- trace.
  1 1-0 primitive trace
   yes
| ?-
```

The format of trace output for one emulator cycle is two lines long. The first line gives the call attempted and the second line gives the result. At the start of the first line the current count of execution steps is given. Following it is the number of active processes separated by a dash from the number of suspended processes. After that the type of process and its goal are given. After writing out the first line, the trace facility waits for the user to type a carriage return before going further. The second line gives the result of one step in the process of attempting to satisfy the current goal. If the goal succeeds, the succeeding value of the goal is also given. An example is the following

```
| ?- trace.
  1 1-0 primitive trace
   yes
| ?-
```
The trace facility distinguishes between the following types of process.

- clause
- literal
- primitive

The types literal and primitive are the types of processes that evaluate user-defined literal goals and system predicate goals. System predicate goals apart from meta-calls never spawn any child processes. By contrast the evaluation of a literal process involves spawning clause processes to evaluate the potential of each clause for that relation to reduce that call. The type clause is used for a process that evaluates the satisfiability of a particular clause. If the clause process commits, it is given the |- prefix. When the guards of clauses are being evaluated to see whether they are satisfiable, an untrusted clause process organises this for each clause. These clause processes are created and scheduled on a demand driven basis. Not all clauses for a relation have a clause process created for them immediately to see whether the clause should be used to reduce the call. The first clause is looked at for a few emulator cycles and if the processes spawned to evaluate its guard succeed, further clause processes to evaluate rival clauses never need to be created. However if that has not happened, a new clause process is spawned to evaluate the guard of the second clause and so on. These clause processes all retain the same literal process parent.

When the emulator commits to a particular clause to reduce a call to the clause’s body, there is no further need for the clause to have a literal parent and the new trusted clause process is promoted on top of the old literal process. Commitment results in the new committed clause process spawning child processes to evaluate the calls in the body of the clause. The general principle is that clause processes spawn children to evaluate the user-defined predicates and the system predicates of the clause. Thus each clause process spawns literal and primitive processes to evaluate the guard of a clause for the given call. These in turn spawn further processes and the process tree grows. Whenever processes succeed or fail, they are removed from the tree. Failure of a process can propagate up the tree, killing off a parent process whose success depends upon the success of the child process and so on upwards. The tree contracts as processes succeed
as well and the whole process tree succeeds when the top level processes that evaluate the top level goals all
succeed. The whole process tree fails when any top level process that evaluates a top level goal fails.

There are several results of executing a process. They depend upon the process’s type. The results of
executing a literal process are as follows.

- busy wait
- spawn
- success
- suspend on children

busy wait signifies that the literal process has created and scheduled another clause process for that literal
goal and is waiting busily for re-execution. spawn signifies that the literal process has created a clause pro-
cess and children guard processes to evaluate the guard of the first clause for a relation. success signifies that
the literal process has an empty guard and an empty body and succeeds immediately. suspend on children
signifies that the literal process cannot create and schedule any more clause processes for that goal and
so suspends upon its clause children.

The results of executing a primitive process are as follows.

- busy wait
- failure
- success
- suspend on input
- suspend on <variables>

The only primitive process that busy waits is the meta-call call/3. It only does this when the current quota
of resources for evaluating that meta-call is used up and the emulator shifts over to emulating some other
processes for a change. failure and success are typical outcomes of evaluation of a primitive process. suspend on input signifies that the primitive has suspended waiting upon input from a file. It will be woken up
when input from that file sufficient to satisfy or fail it becomes available. suspend on <variables> signifies
that the primitive has suspended upon one or more variables. The variables are given. The primitive process
will be woken up if any variable, that it is suspended upon, is instantiated or shared with another variable.
No literal process can suspend upon a variable. Only primitive processes can do so.

The results of executing a clause process are as follows.

- commit
- failure
- success
- suspend on children
- suspend on <variables>
commit signifies that the clause process has committed. failure signifies that the head argument matching requirements of the untrusted clause process cannot be satisfied. success signifies that the clause process has committed and succeeded. suspend on children signifies that the clause process has been descheduled and is waiting until its children succeed or fail. suspend on <variables> signifies that the clause process has been suspended upon goal variables during the head argument matching process and is waiting until they get instantiated further. Trusted clause processes rarely get displayed. They are usually either hidden suspended upon their children, fail offstage or their sole remaining child gets promoted on top of them leaving them no chance to succeed centre stage.

Where there are no more runnable processes but there are processes suspended waiting upon input, the whole evaluation process can go to sleep waiting non-busily for input. This state is signified by the message going to sleep.

Asynchronous input is signified by a message like

* input * get(65,'/dev/tty03').

This happens at the start of an emulator cycle and signifies that the process get/2 waiting for input from the input stream /dev/tty03 has been dealt with and the process scheduled for early execution. Whether it has succeeded or failed is dealt with when the process is run. All pending input is dealt with before any other work is done. Thus several *input* messages may appear together.

Tracing can be done in several modes. These modes are invoked by typing a single command. The available commands can be listed by typing "h" followed by carriage return. The following list of interactive trace commands will be displayed. Each use of a letter command must be terminated by carriage return.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>busy wait active process</td>
</tr>
<tr>
<td>d</td>
<td>depth first execution on active process</td>
</tr>
<tr>
<td>e</td>
<td>end trace</td>
</tr>
<tr>
<td>f</td>
<td>flow execution trace non-interactively</td>
</tr>
<tr>
<td>h</td>
<td>help</td>
</tr>
<tr>
<td>j</td>
<td>jump to given execution step</td>
</tr>
<tr>
<td>l</td>
<td>leap to next call of following name</td>
</tr>
<tr>
<td>n</td>
<td>next leap to prenamed call</td>
</tr>
<tr>
<td>q</td>
<td>quit execution</td>
</tr>
<tr>
<td>s</td>
<td>skip depth first on active process</td>
</tr>
<tr>
<td>t</td>
<td>write process tree to file &quot;tree&quot;</td>
</tr>
</tbody>
</table>

The busy wait command b causes the emulator to busy wait the currently active process. This means that the active point on the scheduling ring is moved one place round. The active process goes to the last
position in the scheduling ring and the next runnable process becomes active. This command provides a simple way of cycling round the runnable process ring without executing anything. It enables a user either to get to the runnable process that the user wants to execute next or to have a look at what is runnable.

The depth-first command d earmarks the active process and any offspring and only executes them in a depth first fashion. Other runnable processes busy wait. The effect is to allow the user to single step the execution of these processes while preventing context switches from making other processes active which interferes with following the execution trace of the nominated process. This execution mode switches itself off when it cannot execute any further. This is indicated to the user.

The end trace command e switches the trace off allowing execution to continue normally.

The flow command f puts the trace into display mode only. The result is that the trace gets displayed without need for interaction. However, once the trace is in display mode only, only the action of turning the trace off and on again will reset it to interactive mode.

The jump command j expects an unsigned integer after it. If this number is larger than the current execution step count, the emulator jumps forward to that execution step. This provides a convenient way of jumping into a computation without necessarily jumping to a particular named call.

The leap command l followed by a sequence of characters unbroken by layout characters will set the leap mechanism to leap until the next call to be made with that relation name.

The next command n has the same effect as l except that it uses a pregiven string value if any have been given. The default string value is the null string. It is useful for repeatedly leaping to the next point at which a named goal is to be executed.

The quit command q causes the emulator to exit at the beginning of the next execution cycle. It provides a tidy way of ending execution. Inside a meta-call it will only result in that meta-call’s goals failing and execution returning to the process tree of which the meta-call is a part.

The skip command s works like the depth-first command d except that it skips tracing until the depth-first execution of the earmarked process and its offspring cannot be carried any further. Execution cannot be carried further either because the skipped from process has failed or succeeded or because the process and its offspring have all suspended. In both cases the final value of the goal is reported. To enable the final value of the goal to be reported without leaving dangling references, all temporary variables in the goal have to be bound to permanent variables in the global heap. This means that V variables always get shared with global variables whatever happens during skipping.

The tree dump command t writes a readable copy of the process tree into a file called "tree" in the current directory along with the active process ring. Processes are indicated by their goal and by the designation PRIM for primitive, LTRL for a literal, C-CL for a committed clause process and U-CL for an uncommitted
clause process. Scheduled processes are marked by a +. A parent process is immediately followed by its children but the children are indented one step further in.

backtrace

In addition to an interactive trace, Kelpie also has a post-mortem execution reporting facility called a "backtrace". The backtrace can be toggled on and off.

| ?- backtrace.

<table>
<thead>
<tr>
<th>Execution type</th>
<th>Number</th>
<th>Memory Use</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>clause searches</td>
<td>0</td>
<td>active process peak</td>
<td>1</td>
</tr>
<tr>
<td>context switches</td>
<td>0</td>
<td>data records allocated</td>
<td>0</td>
</tr>
<tr>
<td>execution cycles</td>
<td>1</td>
<td>processes allocated</td>
<td>1</td>
</tr>
<tr>
<td>reductions</td>
<td>0</td>
<td>suspended process peak</td>
<td>0</td>
</tr>
<tr>
<td>undefined calls</td>
<td>0</td>
<td>variable suspenses peak</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process type</th>
<th>Number</th>
<th>Outcomes</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>literals</td>
<td>0</td>
<td>successes</td>
<td>1</td>
</tr>
<tr>
<td>primitives</td>
<td>1</td>
<td>failures</td>
<td>0</td>
</tr>
<tr>
<td>trusted clauses</td>
<td>0</td>
<td>parent suspensions</td>
<td>0</td>
</tr>
<tr>
<td>untrust clauses</td>
<td>0</td>
<td>variable suspensions</td>
<td>0</td>
</tr>
</tbody>
</table>

execution time 0.34 seconds

| yes |
| ?- |

The backtrace reports on peak memory usage, on types of execution steps, on the number of reductions performed, on undefined literal calls and gives cpu time devoted to execution.

Error Handling

The Kelpie system has its own operating system signal handler. The operating system signal handler catches the following signals and performs the following actions.
<table>
<thead>
<tr>
<th>Signal</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus error</td>
<td>halt Kelpie</td>
</tr>
<tr>
<td>Continue signal</td>
<td>give warning</td>
</tr>
<tr>
<td>Floating point exception</td>
<td>give warning</td>
</tr>
<tr>
<td>Hang up signal</td>
<td>give warning</td>
</tr>
<tr>
<td>Input/output signal</td>
<td>set asynchronous input flag</td>
</tr>
<tr>
<td>Interrupt signal</td>
<td>stop emulator at start of next cycle</td>
</tr>
<tr>
<td>Interrupt signals</td>
<td>force restart of emulator at top level</td>
</tr>
<tr>
<td>Segmentation fault</td>
<td>halt Kelpie</td>
</tr>
<tr>
<td>System call invalid</td>
<td>give warning</td>
</tr>
<tr>
<td>Software termination</td>
<td>give warning</td>
</tr>
<tr>
<td>Stop wait on input</td>
<td>halt Kelpie</td>
</tr>
<tr>
<td>Stop wait on output</td>
<td>halt Kelpie</td>
</tr>
</tbody>
</table>

Internal errors are caught by Kelpie’s own error handler. The general rule is that errors are allowed to propagate upwards to the emulator level or if generated during reading or consulting to an appropriate lower level. They can be of the following kinds.

- Execution error
- Invalid argument error
- Parsing error
- Semantic error
- Start up error
- Syntax error
- Syntax warning
- System error
- System warning
- Tokenising warning

System error messages are fatal. Most have unexceptional causes like exceeding memory space allocations. A few system errors can only be due to implementation bugs and are the following.

- Encountered unrecognised variable
- Evaluation error
- Not a recognised system call
- Unexpected cell tag
- Unlinked complex term encountered
- Unrecognised token

If any of these errors occur, they should be reported with enough information to be able to reproduce the error.

**Acknowledgements**

The author wishes to acknowledge the financial support received from the Science and Engineering Research Council under the Alvey initiative, help from members of Imperial College’s Parlog group, the support of his wife and colleagues at Heriot-Watt University and his project’s industrial partner ICL.
References

Agha 1986.

Ait-Kaci & Nasr 1986.


Backus 1978.


Bancilhon & Ramakrishnan 1986.

Barbuti & Martelli 1986.


Bocca 1986.


Butler et al 1986.


Chang et al 1985.


Chikayama 1983.

Church 1936.

Clark & Tamlund 1977.
Clark 1978.

Clark & McCabe 1979.

Clark 1979.
   K.L. Clark, "Predicate Logic as a Computational Formalism," Research Report DOC 79/59, Depart-

Clark & Gregory 1981.
   ACM Conference on Functional Programming Languages and Computer Architectures, pp. 171-178,

Clark & McCabe 1982.
   Intelligence, ed. J.E. Hayes, D. Michie and Y-H. Pao, Vol. 10, pp. 455-470, Ellis Horwood, Chich-

Clark & Gregory 1984.
   84/15, Department of Computer Science, Imperial College, London, (April 1984).

Clark & Gregory 1985.

Clark & Gregory 1986.
   K.L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," ACM Transactions on Pro-

Clark & Gregory 1987.

Clark 1988.
   K.L. Clark, "Logic Programming Schemes," in Proceedings of the Fifth Generation Computer Sys-
   tems conference, Tokyo, (November 1988).

Clocksin & Mellish 1981.

   W.F. Clocksin, "Design and Simulation of a Sequential Prolog Machine," New Generation Comput-
   ing, pp. 101-120, OHMSHA Ltd. and Springer-Verlag, (1985).

Codd 1970.
   E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the

Codish & Shapiro 1986.
   M. Codish and E. Shapiro, "Compiling Or-parallelism into AND-parallelism," in Lecture Notes in

   C. Codognet, P. Codognet, and G. File, "Yet Another Intelligent Backtracking Method ' - in Logic
   Programming, Proceedings of the Fifth International Conference & Symposium on Logic Program-
Colmerauer et al 1983.

Colmerauer 1987.

Colomb 1989.

Conery & Kibler 1985.

Conery 1986.

Conery 1987.


Crammond 1986.


Cristian 1982.


Date 1986.

D. DeGroot, "Restricted AND-Parallelism," in Proceedings of the International Conference on Fifth

van Emden & Kowalski 1976.

FBI 88.

Forsyth 1989.

Foster 1987.

Foster & Taylor 1987.

Foster 1988.

Frege 1879.

Fuchi 1983.

Fuchi & Furukawa 1986.


Gallaire 1983.


Gee et al 1987.

van Gelder 1987.

Gentzen 1969.

Godel 1930.

Godel 1931.

Goldberg & Robson 1983.
A. Goldberg and D. Robson, Smalltalk-80: the Language and its Implementation, Addison-Wesley, (1983).

Goodenough 1975.


Hammond & Sergot 1983.

Haridi & Ciepielewski 1983.

Harman 1968.


Hayes 1979.

Henderson 1982.

Hermenegildo 1986.


Hoare 1978.


Ichiyoshi et al 1987.

Itoh 1986.


H. Itoh, H. Monoi, and H. Seki, "Knowledge Base System towards Logic Programming and Parallel


Kale 1987a.

Kale 1987b.


Kemp & Topor 1988.


Kowalski 1974.
Kowalski 1979.

Kowalski 1981.

Kowalski 1982.

Kripke 1969.

Levy 1986a.

Levy 1986b.

J. Levy and E. Shapiro, "Translation of Safe GHC and Safe Concurrent Prolog to FCP," Technical Report CS87-08, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, (June 1987).

Lin et al 1986.


Lloyd 1984.

Lloyd & Topor 1986.

P.A. Massey, A Transaction System for a Multi-user Prolog Database System, Department of Computer Science, Heriot-Watt University, (November 1988).


Michaelson & Gregory 1990.

Minsky 1981.

Miyazaki et al 1986.

Moss 1986.


Murakami et al 1983.

Mylopoulos 1986.

Naish 1986.

Naish et al 1987.


Ng & Leung 1988.

Nicolas & Gallaire 1978.


Pereira et al 1986.

Pollard 1981.

Pyle 1981.


Reiter 1984.


Robinson 1965.

Roussel 1975.

Ryle 1949.

Salvini 1989.

Saraswat 1987b.

Saraswat 1987a.


Shapiro 1983.

Shapiro & Takeuchi 1983.

Shapiro 1986.

Shapiro 1987.

Shapiro 1989.


Sterling 1984.

Sterling & Shapiro 1986.


Subrahmanyan 1985.


Tamaki & Sato 1986.
Tamaki 1987.


Taylor 1988


Thom and Zobel 1986.


Ueda 1985.

Ueda 1985.

Ueda 1986b.

Ueda 1986a.

Ueda 1986c.

Ueda 1987.


P J. Voila, "Types of Trilogy," in Logic Programming, Proceedings of the Fifth International


Warren 1983.


Warren 1990.


Whitehead & Russell 1910.

Wiederhold 1986.


Williams, Kong & Chen 1988.

Wong & Williams 1988.

Woo & Choe 1986.


R. Yang, Programming in Andorra-I, Department of Computer Science, University of Bristol, Bristol, (September 1988).


Yokota & Itch 1986.


Zloof 1977.