PARALLEL EXECUTION

OF

HORN CLAUSE PROGRAMS

by

George H. Pollard

A Thesis Submitted

for the Degree

of

DOCTOR OF PHILOSOPHY

of the

University of London

Imperial College of
Science & Technology

1981

ABSTRACT

The thesis is primarily concerned with investigating the potential for concurrent execution inherent in Horn clause programs and how that potential might be realised.

The early chapters give the background and introduce the concepts which underpin the use of Horn clauses for the expression of computer programs. A description which outlines how such programs are interpreted on conventional von Neumann architectures then follows. The preliminary part of the thesis closes with a chapter that describes the forms of parallelism intrinsic to Horn clause programs and how such parallelism might be exploited to good effect.

Our principal research contribution then follows. This takes the form of two schemes for the parallel execution of Horn clause programs.

The first, the Or-parallel scheme, is an extension of the conventional backtracking scheme and our coverage of it is quite comprehensive, extending to the description of computer architectures which might be used to implement it.

The second, the And-or scheme, is more ambitious insofar as it has the potential to exploit all the parallelism implicit in Horn clause programs. This scheme is a radical departure from conventional approaches to Horn clause program execution and our presentation concentrates on its design rather than on its possible implementation.

To My Parents

ACKNOWLEDGEMENT

I would like to place on record my gratitude to British Telecom who, as my employers, have recognised the potential of Logic Programming and have given me the opportunity to contribute towards the research needed to fulfil that potential. Foremost amongst those at BT to whom personal thanks are due is my colleague, Jonathan Porter, who supported my application for the scholarship in the first place and has taken an active interest in my researches since.

I would also like to thank my family for all their help and encouragement. My parents have assisted my career through many sacrifices of their own and their interest in my recent research activities has not gone by unappreciated. I also would like to thank Dr R.C.H. Tanner who, as a family friend, has keenly followed my progress and has in many ways assisted it. Her help was invaluable in establishing the foundations of my career.

Thanks are due to my wife Lorna and children Andrew, David and Jonathan for having put up so equably with the occasional disadvantage dictated by the needs of intensive research.

Unquestionably, my deepest thanks go to Bob Kowalski, my supervisor, without whose insight, assistance and guidance (not to mention pioneering work) I would not have been able to undertake this project. Bob possesses the all too rare quality of being able to view an issue from many standpoints, a quality which, when coupled with his

ability to communicate so clearly, has been invaluable to the research reported here. On many occasions, a discussion with him has left me greatly enlightened and encouraged by his comments and enthusiasm. I feel a deep sense of privilege for having had the opportunity to study under him.

Finally, I would like to thank Frank McCabe for his valuable comments on earlier drafts of this thesis and, more generally, to all my colleagues both at Imperial College and British Telecom for the assistance they have given me.

CONTENTS

1.0 CHAPTER 1: Introduction 9
1.1 Background 9
1.1.1 Programming 9
1.1.2 Data Base Versatility 11
1.1.3 Parallelism 12
1.2 Statement of Thesis 13
1.2.1 Objectives of Research 14
1.3 Preview of Contents 14

2.0 CHAPTER 2: Horn clause Programming 16
2.1 Syntax 16
2.2 Semantics 20
2.2.1 Declarative Semantics 20
2.2.2 Operational Semantics 21
2.3 Prolog 25
2.3.1 Search Tree 26
2.3.2 Control 28
2.3.3 Negation as Failure 28
2.3.4 Built-in Predicates 30

3.0 CHAPTER 3: A Basic Horn Clause Interpreter 33
3.1 Introduction 33
3.2 Outline Requirements of the Interpreter 33
3.3 Abstract Interpreter 34
3.3.1 Search Strategy 34
3.3.2 Selection Strategy 35
3.3.3 Variable Naming 36
3.3.4 Structure-sharing 37
3.3.5 Bindings 37
3.4 Implementation 39
3.4.1 Goal List 41
3.4.2 Description of Processing 42

4.0 CHAPTER 4: Parallelism 43
4.1 Introduction 43

4.2 Potential for Parallelism 43
4.3 Or-parallelism 45
4.3.1 Database Applications 45
4.3.2 Functional Problems 46
4.3.3 Negative Literals 48
4.3.4 Implicative Literals 48
4.4 And-parallelism 51
4.4.1 Independent Subgoals 51
4.4.2 Pipelining 52
4.4.3 Early Detection of Failure 58
4.5 Regulation of Parallelism 61
4.6 Introduction to the Schemes 63

5.0 CHAPTER 5: Or-parallel Proof Procedure 65
5.1 Introduction 65
5.2 Basic Requirement 65
5.3 Implementation Decisions 65
5.4 A Naive Model of the Implementation 68
5.5 A More Practical Model of the Implementation 69
5.5.1 Representation of Bindings 69
5.5.2 Interrelation of Activation Records 72
5.5.3 Processes and Messages 73
5.5.4 A Simple Computation 74
5.5.5 Main Processes 75
5.5.6 Registration 78
5.5.7 Solution Extraction 82
5.5.8 Branch Names 83
5.5.8.1 N-ary Branch Naming Scheme 83
5.5.8.2 Binary Branch Naming Scheme 85
5.5.8.3 Integration of N-ary and Binary Branch Naming Schemes 86
5.5.8.4 Pre-allocation of Branch Names 87
5.6 Register Structure and Manipulation 89
5.6.1 Structure 89
5.6.2 Insertions 91
5.6.3 Deletions 95
5.7 Database Applications 96
5.8 Architecture 100
5.8.1 Requirements 100
5.8.2 Memory 100

5.8.3 Packets 102
5.8.4 Distribution of Work 103
5.8.5 Message Communication 104
5.8.5.1 Internal PE Structure 108
5.8.6 Processor-Memory Connection 110
5.8.7 Storage Management. 113
5.8.8 Process Control 114
5.8.9 Modifications to the Basic Scheme 116
5.8.9.1 Specialist PE's 116
5.8.9.2 Search Engine 118
5.8.10 Regulation of Parallelism 118
5.9 Assessment 120
5.9.1 Level of Parallelism 120
5.9.2 Low Degree of Concurrency 121
5.9.3 High Degree of Concurrency 123

6.0 CHAPTER 6: And-or Proof Procedure 126
6.1 Introduction 126
6.2 And-or Tree 127
6.3 Introductory Example 128
6.4 The Basic Scheme 135
6.4.1 Reconciliation 135
6.4.2 Solutions 137
6.4.3 Registration 139
6.4.4 Scope 140
6.4.5 Conjointness and Disjointness 141
6.4.6 Filtering and Pruning of the And-or Tree 146
6.4.6.1 Scope Subsumption 147
6.4.6.2 Promotion 148
6.4.6.3 Reduction 150
6.4.6.4 Pruning 150
6.5 Controlling the Concurrency 158
6.5.1 Dynamic Control of Activity 158
6.5.2 Suppression of Unproductive Parallelism 159
6.5.2.1 Language Modification 160
6.5.2.2 Modification to Proof Procedure 161
6.5.2.3 Relationship to Or-parallel Proof Procedure 167
6.5.3 Networks 168
6.6 Implementation Considerations 172

6.6.1 Structure-sharing 172
6.6.2 Unification 173
6.6.2.1 Choice of Bindings and Timing Considerations 174
6.6.2.2 Parallel Unification 177
6.6.3 Devolution of Processing 178
6.6.3.1 Ownership of Unifiers and Filters 178
6.6.3.2 Filter Subsumption Check. 179
6.6.3.3 Scope Subsumption 180
6.6.4 Determination of Conjointness/Disjointness 182
6.6.5 Registers 183
6.6.5.1 Registration 183
6.6.5.2 Reconciliation 184
6.6.6 Filter Incorporation 185
6.6.6.1 Promote and Reduce Phase 186
6.6.6.2 Filter Implementation Phase 187
6.6.6.3 Weaknesses in Filter Manipulation 189
6.7 Assessment 193
6.7.1 Degree of Parallelism 193
6.7.2 Termination 193
6.7.3 Control 194
6.7.4 Filter Manipulation 195
6.7.5 Architecture 195

7.0 CHAPTER 7: Conclusion 197
7.1 Related Research 197
7.2 Future Research 207
7.2.1 Or-parallel Proof Procedure 207
7.2.2 And-or Proof Procedure 208

8.0 References 209

9.0 APPENDIX: Details of Basic Interpreter 214

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

It is perhaps surprising that an industry which likes to surround itself in the aura of logical thought should be taking so long to discover logic itself.

The academic world has been actively investigating Logic Programming [25] since 1972 and recent years have seen a great increase in the level of interest shown in this area of computing science.

The computing industry, on the other hand, is distinctly lagging in this field yet as is so often the case, it is the industry itself which has the most to gain. The evidence that Logic Programming is of practical use has been available for some time [18] but to date there has been no large-scale application of it. Perhaps this is not surprising in view of the departure from convention that this formalism entails.

Much research effort is being applied to the development of Logic Programming; some, such as the research reported here, being supplied by non-academic bodies. In this context, it is worth pointing out that the largely unpublicised Japanese Government and Computer Industries coordinated research program into fifth-generation computers [45] is based on the use of Logic Programming and kindred formalisms.

Below, we outline some of the features of programming in logic and the potential for improvement and advancement that this formalism offers.

1.1.1 Programming

It is generally accepted that automatic computation is plagued by

all sorts of problems stemming from the use of procedural languages. By 'procedural languages' we mean languages whose semantics are given in terms of the state of some abstract (occasionally real!) machine.

The problems intended for automatic computation are expressed in quite a different manner. As a rule, they are stated informally in a system specification, by and large in a style oriented towards the real world. It is then the programmer's task to take such an informal specification and translate his interpretation of it into a form which is not human-oriented (and consequently difficult to comprehend).

It is hardly surprising that at one and the same time, the conventional approach produces programs that are difficult to understand (and hence modify) and which are also wrong - insofar that they seldom produce the intended results and in any case promote little faith in their 'correctness'.

It is true that certain advances, notably structured programming, have tended to mitigate the worst excesses of this approach but they have not addressed the central question of presenting problems to the computation process in a manner truly understandable by both man and machine.

To overcome these shortcomings, the machine must, at some level, be made to understand naturally expressed problems.

Prolog [41], [38], [43] is a significant advance in this respect, for although it is by no means a natural language, it is far closer related to human thought and expression than procedural languages are. The justification for this statement is that Prolog is essentially the Horn clause subset of Standard Form Logic [24], a formalism whose roots go back over two thousand years and whose development ever since has been motivated by the need to assist the comprehension of human thought. Procedural computer languages, on the other hand, exist solely for the purposes of automated computation.

Much Logic Programming research effort is being applied to the area of program transformation [8], [23]. The eventual aim here is to

allow the user to express the problem domain at a still higher level - perhaps a level equivalent to the Standard Form of Logic - and then to transform the supplied specification to a (logically equivalent) form that also exhibits good computational behaviour.

Similar research is being undertaken for functional programming languages (e.g. [3]) but the Logic Programming approach has the unique characteristic that only one formalism is involved.

Clearly, significant advances are offered by the success of these investigations.

1.1.2 Data Base Versatility

Much of the interest being shown in Prolog stems from its properties in the context of database application [19].

It is possible to interpret any relational database (as described by Codd [9]) as a Prolog program but the converse is not true - i.e. Prolog is more general. It is reasonable to expect therefore that much of the interest being shown towards relational databases will find expression as an interest in Prolog.

In Prolog, there is actually no distinction between what is conventionally considered program and data. A 'program' is merely an implicit manifestation of data insofar as its sole purpose is the computation of that data (this point will be amplified later).

The capability of being able to express both 'program' and 'data' in a single formalism is a truly remarkable one but even so, it does not exhaust the expressive versatility of Prolog. Because Prolog is logic, aspects of computation associated with logic may be directly implemented in Prolog. An example appropriate to the area of databases is that of integrity constraints.

Continuing this theme of Prolog's versatility, we mention its use as a meta-language. Briefly, a meta-language is one which reasons about another language, the object-language - i.e. the objects represented in the meta-language are statements of the object-language. Both may be implemented in Prolog. An illustration of the use of meta-language might be that concerning the modification of a database. Here, the object being manipulated in the meta-language is the database itself. The integration of object- and meta- level programming in an elegant way is under active investigation [1], [44] and offers the dual benefits of safer and more powerful programming.

1.1.3 Parallelism

Recent advances in micro-electronics, particularly in the area of VLSI fabrication, have served to reduce the cost of computer hardware. The trend is a continuing one and affects the level at which theoretical improvements to computation become cost-effective practicalities.

One such area is that of parallelism [42]. Parallelism, in the form of pipelining within instruction cycles, has for some time been a standard feature of many conventional machines. However, such pipelining depends on the identification of separate components of some action and the execution of those sub-actions on an input stream - neighbouring components operating on neighbouring items in the stream. This form of parallelism is not readily extensible and so cannot take proper advantage of the opportunities offered by incoming technology.

Parallelism at the higher level of concurrency of program instruction execution is a more recent innovation [22], [46] and holds some promise.

However, this approach to exploiting future technology is not ideal either. For one thing, it is the programmer who decides whether to incorporate such concurrency or not and in this way, his influence over the computation process is extended rather than diminished.

Of more immediate concern here is the criticism that this form of concurrency is not sufficiently general and therefore does not hold maximal potential for such exploitation. Concurrency in these languages tends to be rather coarse-grained because it is the programmer's responsibility to identify 'processes' that might run concurrently.

Perhaps a more serious criticism of such concurrency follows on from the earlier discussion of semantics. The semantics of fully transparent concurrent computation has not been defined - in all probability cannot be defined - in terms of the effects of sequential transitions on the state of a machine.

Prolog and other languages, notably functional languages, whose semantics may be given in human terms rather than in terms of machine state transitions, do not suffer the above limitations. So the single step necessary to implement concurrency for such languages is to supply a mechanism that conforms to those semantics and yet is concurrent. The concurrency thus conferred could be made transparent (i.e. not of the programmer's concern), a significant point in its favour.

1.2 STATEMENT OF THESIS

1. Concurrent computation is a desirable way to exploit VLSI.

2. Prolog is a suitable language for the advancement of concurrent computation.

3. Concurrent implementations will serve to make Prolog a more attractive and hence widespread language.

4. Increased use of Prolog will serve to improve the general standard of programming.

The research reported in this document is primarily concerned with item 2.

1.2.1 Objectives of Research

The major area of original research reported herein is concerned with the investigation of alternative approaches to the implementation of a concurrent execution mechanism for Prolog programs.

Throughout this research, the criterion of transparency has been regarded as a most desirable property. The few departures from this ideal that our proposals make are necessary solely in the interests of expediency and it is confidently expected that in due course, such expediency will not be necessary and this criterion will be fully realised.

The schemes proposed are quite general and make no assumptions about the nature of the programs to which they might be applied. They contrast in this respect with other schemes (e.g. [5]) which impose certain restrictions on the user's program.

1.3 PREVIEW OF CONTENTS

The thesis may be regarded as consisting of three parts.

The first part is concerned with well-established aspects of the use of logic for programming.

Chapter 2 deals with the theory underlying Horn clause programming and includes a section on the programming language Prolog. Chapter 3 continues in this vein by describing an idealised conventional implementation of a basic Prolog interpreter. This account will be of some use in later parts of the thesis because it affords the possibility of

comparing and contrasting certain aspects of the new approaches with those found in conventional implementations.

Chapter 4 makes precise the notion of concurrency as applied to the execution of Horn clause programs and investigates the various ways in which such concurrency may be used to advantage.

The second part of the thesis comprises our principal contribution of original work. It consists of two chapters, each describing a parallel execution mechanism for Horn clause programs.

The first of these, Chapter 5, describes the Or-parallel Proof Procedure, a scheme which exploits just one form of parallelism. We consider it feasible to implement this scheme by means of technology now available. Because of this and the fact that the abstract model of the proof procedure is quite simple, our exposition here has an implementation bias to it and includes, for example, a discussion on architectures that might be used to realise the scheme.

Chapter 6 describes the And-or Proof Procedure, an execution mechanism with the potential to exploit, we believe, all the parallelism implicit in Horn clause programs. The scheme, as one might expect, is more complex and consequently our description is pitched at a higher level.

The final part of the thesis, Chapter 7, is more general and relates our researches to those being undertaken by others. It also, in conjunction with chapters 5 and 6, identifies areas of further research.

CHAPTER 2: HORN CLAUSE PROGRAMMING

There are several excellent introductions to Horn clause programming, notably Kowalski's book [28]. In view of this, it is not appropriate to devote a great deal of space here to such an introduction, and the interested reader is earnestly advised to refer to the above book.

However, we do wish the thesis to be as self-contained as possible and in view of this, and the fact that there is a certain variation in terminology throughout the literature, we will be quite specific and include here the background material necessary to enable the thesis as a whole to be understood.

2.1 SYNTAX

A Horn clause logic program is a set of **clauses**.

The building blocks of clauses are atoms. An **atom** expresses a relationship between the individuals that appear in it (or a property of an individual if just one appears).

Examples of atoms are

Andrew is the brother of David
0 is a number
Jonathan likes x

where the names of the relationships are underlined and the names of the individuals are not.

We will for the most part find it convenient to use prefix representations of atoms thus

Brother(Andrew, David)

Number(0)

Likes(Jonathan, x)

There are two types of clauses in a logic program: assertions and implications.

An **assertion** is comprised of a single atom. It is a statement of fact and is read as holding unconditionally.

An **implication** is a statement of the form

B is implied by A_1 and A_2 and ... and A_n

written

$B \leftarrow A_1 \ \& \ A_2 \ \& \ \ldots \ \& \ A_n$

where A_1, A_2, ..., A_n and B are all atoms. B is termed the **consequent** atom of the implication and A_1, A_2, ..., A_n its **antecedent** atoms.

An example of an implication is

Happy(John) <- Friend-of(John, Mary) & Friend-of(John, Jane)

No significance is attached to the ordering of the atoms in the antecedent of an implication. Note that an assertion is simply the special case of an implication with no antecedent atoms.

The consequent atom of a clause will sometimes be termed its **head**; the conjunction of its antecedent atoms, its **body**.

An atom is made up of a predicate symbol and one or more terms. As indicated above, we will tend to use the prefix representation of atoms P(....), although occasionally we will opt for the infix form when only two terms are involved e.g. x = A.

A term is a **constant**, a **variable** or a **functor**.

John and 0 are examples of constants. Each use of a constant in the program refers to the same individual.

All variables appearing in a clause are interpreted as being universally quantified outside the clause. For example the clause

```
Parents-of(x, y, z) <- Mother-of(x, y) & Father-of(x, z)
```

is interpreted as

```
For all x,y,z (Parents-of(x, y, z)
                <- Mother-of(x, y) & Father-of(x, z) ).
```

For those variables which only appear in the body of a clause, an equivalent interpretation is obtained by existentially quantifying the variables concerned immediately outside the body.

For example,

```
For all x,y,z (Grandparent(x, y) <- Parent(x, z) & Parent(z, y) )
```

is read alternatively as

```
For all x,y (Grandparent(x, y)
              <- There exists z (Parent(x, z) & Parent(z, y) ) ).
```

Variables, unlike constants, are only local to the clause in which they appear i.e. there is no special relationship between occurrences of the same textual variable in two different clauses. Conversely, two clauses identical in every respect except for a one-to-one mapping of variables are in fact read as being identical. Thus

```
Grandparent(x, y) <- Parent(x, z) & Parent(z, y)
```

and

```
Grandparent(u, v) <- Parent(u, w) & Parent(w, v)
```

are clauses with identical meaning and are termed **variants** of one another.

The two above properties of variables are consequences of the 'universally quantified' interpretation placed on variables in a Horn clauses program.

Functors consist of a **function symbol** and one or more terms. (An equivalent alternative definition dispenses with constants and allows functors with no terms in their place.)

Functors are analogous to data structures in conventional programming. Examples of functors are

```
date(1, Jan, 82)
.(x, NIL)
x.NIL
```

The function symbols in the above examples are 'date', '.' and '.' respectively. As with atoms, functors may be written in prefix or infix form. The terms in the prefix form are separated by commas.

The third example is the infix equivalent of the second. The conventional usage of the '.' functor - and that adopted throughout this thesis - is in the representation of lists. Both examples name the list whose first item is x and whose remainder is the list NIL (a constant conventionally representing the empty list).

Notice that the definition of term is recursive. Thus .(A, .(B, x)) names the list whose first two items are A and B and whose tail is the list x. The equivalent infix form is A.(B.x). By assuming that '.' is associative to the right, we will dispense with the brackets altogether and use A.B.x to represent such a term.

Functors and atoms share the same syntax but are distinguished by the context of their occurrence in the clause. We will distinguish between variables and constants by using names beginning with a lower case letter to represent variables.

2.2 SEMANTICS

Two principal semantics may be ascribed to Horn clause logic:- the declarative semantics (human oriented) and the operational semantics (machine oriented). It is the equivalence of the two that imputes to Horn clause programs the dual aspects of being human-oriented and machine executable.

2.2.1 Declarative Semantics

In Horn clause programs, no 'meaning' as such attaches to the symbols of the program; the only 'meaning' possible is that which can be inferred from the set of clauses that make up the program.

The semantics of Horn clause logic may be given in terms of logical implication, which in turn is normally given in terms of the notions of interpretation and inconsistency.

Although the declarative semantics are important in theory, their equivalence to the operational semantics makes it possible to dispense with their exposition here and rely instead on a detailed description of the operational semantics. A rigorous description of the declarative semantics of clausal logic and its equivalence to the operational semantics is given in [15].

We contend that people have a sufficiently intuitive idea of 'truth' and that when a user is engaged in practical programming, it is good enough for him to write clauses which are 'true' statements about the intended meanings of the symbols used in the program. A choice of meaningful symbols is therefore essential in practice.

2.2.2 Operational Semantics

To be activated, a Horn clause program must be presented with a goal statement. A goal statement is a clause of the form

$$\leftarrow G_1 \ \& \ G_2 \ \& \ \ldots \ \& \ G_n$$

where G_1, G_2, ..., G_n are atoms called goals. In the declarative reading, a goal statement is taken as the denial

> "for all v_1, v_2, ..., v_m (the variables appearing in the terms within the atoms), it is not the case that G_1 & G_2 & ... & G_n is logically implied by the program".

It is possible to show that if the set of clauses given by the union of the program and the goal statement is inconsistent, the conjunction (G_1 & G_2 &... & G_n) is logically implied by the program, for some instance of its variables.

Resolution

The operational semantics of Horn clauses are essentially given in terms of resolution [39]. Although resolution is a quite general inference mechanism for clausal form, this account of it is given in terms of top-down (goal-directed) computation for Horn clauses, the form in which it is almost universally used in practice.

Let the given goal statement be

$$\leftarrow A_1 \ \& \ A_2 \ \& \ \ldots \ \& \ A_n$$

and select one of the atoms A_i, whose predicate symbol is named P.

Select from the program a clause

$$B \leftarrow B_1 \ \& \ B_2 \ \& \ \ldots \ \& \ B_m$$

whose head B also has the predicate symbol P.

A step of resolution is the process whereby the goal atom, A_i, and the clause head, B, are matched with most general unifier S and a new goal statement is derived by application of S to

$$\leftarrow A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_{i-1} \,\&\, B_1 \,\&\, \ldots \,\&\, B_m \,\&\, A_{i+1} \,\&\, \ldots \,\&\, A_n$$

- i.e. the original goal statement with the selected goal atom replaced by the body of the selected clause. (Variables must, if necessary, be renamed so that no variable appears in both the goal statement and selected clause.)

Two atoms, A and B, may be **matched** or **unified** with **most general unifier** S if S is a substitution whose application to A and B results in a common instance of A and B which is most general.

A **substitution** S is a set of **substitution components** (or **bindings**), $\{v_1/t_1,\ v_2/t_2,\ \ldots,\ v_k/t_k\}$, where each component is a "variable/term" pair and $v_i=v_j$ only if i=j.

The **application** of the substitution S to an expression A is the process of replacing each occurrence of every variable v in A by the term t, for each component v/t in S. The expression (A)S produced by this means is the **instance** of A determined by S.

C is a **common instance** of A and B if it is an instance of A and an instance of B determined by the same substitution. C is **most general** if every other common instance of A and B is an instance of C.

Derivation and Refutation

A resolution step produces a new goal statement (G') from the old (G) or the empty clause in the special case when only one goal appears in G and the selected clause is an assertion.

The empty clause is a special clause which has no head or body and has the unique property of being self-contradictory.

Given a program P and initial goal statement G_1, a sequence of goal statements G_1, G_2, ..., G_n produced by resolution steps is a **derivation** of G_n from S, where S is the union of P and G_1.

If G_n is the empty clause, this derivation is a **refutation** of S.

It may be shown that if there exists a refutation of a set of clauses S then S is inconsistent [39]. In the context of goal-directed computation, this is equivalent to stating that an instance of the conjunction making up the body of the goal statement (the goal conjunction) is logically implied by the program. The instance is the one determined by applying to the goal conjunction the union of all most general unifiers produced in the course of refutation.

The exhibition of such a refutation is a proof of the goal conjunction which forms the body of G.

The Procedural Interpretation

Kowalski's procedural interpretation of Horn clause programs [25] is a convenient way of expressing the mechanics of resolution inference in conventional computing terms.

In the procedural interpretation, the goal statement <- G_1 & G_2 & ... & G_n is interpreted as a request to find an instance of the variables that conjointly (simultaneously) solve the goals G_1, G_2, ..., G_n.

Procedure Calls and Definitions

The goal statement <- G_1 & G_2 & ... & G_n is interpreted as a set of n procedure calls, the arguments of each call being the terms within the respective atom. The calls must be solved conjointly in order to constitute a solution of the goal statement. No significance is attached to the order in which goals are selected for solution [20].

A clause whose head predicate symbol is P is called a **procedure definition for P.**

The set of procedure definitions for P is called the **procedure set** for P. The set of procedure sets thus partitions the program.

In the procedural interpretation, the definitions in the procedure set for P are interpreted as alternative ways to solve calls whose associated symbol is P. The terms of the consequent atom are interpreted as procedure head arguments and serve to identify the procedure calls which the definition can be used to solve.

Activation and Invocation

In the procedural interpretation, calls are **activated** and procedure definitions are **invoked** in response.

Activation consists of selecting as procedure call, a goal from the set that makes up the goal statement. The rule governing which call is chosen is known as the **selection strategy.**

Invocation entails the selection of a procedure definition from the procedure set appropriate to the selected goal and the use of that definition with a view to advancing the computation. The rule governing which definition is chosen is known as the **search strategy.**

Computation Steps

In the procedural interpretation, a resolution step is considered a step of computation. Computation terminates when the empty clause is derived.

Transmission of Data

The act of matching a procedure call with the head of a procedure definition is viewed in the procedural interpretation as the transmission of data between the call and definition. The data transmitted is to be found in the associated most general unifier.

Let S be a most general unifier produced in a computation step. The components of S may be classified according to the criterion of whether the component's variable originates from the goal statement or whether it comes from the selected procedure definition. The set S may thus be partitioned: S = S-in U S-out, where S-in is the set of components for procedure definition variables and S-out is the corresponding set for goal variables.

Application of S-in to the atoms of the goal statement has no effect; likewise, application of S-out to the atoms originating from the body of the definition also has no effect. Therefore, that part of the computational step which stipulates the application of S to

$$(A_1 \ \& \ A_2 \ \& \ \ldots \ \& \ A_{i-1} \ \& \ B_1 \ \& \ \ldots \ \& \ B_m \ \& \ A_{i+1} \ \& \ \ldots \ \& \ A_n)$$

may be viewed as an application of S-in to $(B_1 \ \& \ B_2 \ \& \ \ldots \ \& \ B_m)$ and an application of S-out to $(A_1 \ \& \ A_2 \ \& \ \ldots A_{i-1} \ \& \ A_{i+1} \ \& \ \ldots \ \& \ A_n)$.

If now this computational step forms part of a successful refutation, the procedural interpretation views the application of S-in to $(B_1 \ \& \ B_2 \ \& \ \ldots \ \& \ B_m)$ as transmission of input data to the invoked procedure definition and the application of S-out to $(A_1 \ \& \ A_2 \ \& \ \ldots A_{i-1} \ \& \ A_{i+1} \ \& \ \ldots \ \& \ A_n)$ as the transmission of output data from the invoked procedure definition.

2.3 PROLOG

The computer language Prolog, first implemented in 1972 by Colmerauer and his colleagues at Aix-Marseille [10], is essentially the language of Horn clause logic and its implementation is closely based on Kowalski's procedural interpretation.

The two uncommitted aspects of the procedural interpretation, namely the selection strategy and the search strategy, are usually implemented in the following particularly simple ways.

The selection strategy is left-right, last-in-first-out i.e. given the goal statement $\leftarrow A_1 \,\&\, A_2 \,\&\ldots\, \&\, A_n$, the call A_1 is selected and if the clause $B \leftarrow B_1 \,\&\, B_2 \,\&\, \ldots \,\&\, B_m$ is invoked in response to it, the new goal statement (assuming successful unification) will be

$$\leftarrow B_1' \,\&\, B_2' \,\&\ldots\, \&\, B_m' \,\&\, A_2' \,\&\, \ldots \,\&\, A_n'$$

(primes indicating modifications resulting from the application of the most general unifier), from which the next call selected will be B_1' etc..

For the purposes of the search strategy, procedure sets are regarded as procedure lists. The list order is that given by the textual ordering of the clauses making up the procedure set.

Initially, the clause invoked in response to some given call is the first in the list specified for the call's predicate symbol. In the event of a matching failure or possibly a subsequent return to this point because of later failures, the second clause, if one exists, will be tried etc..

In the event of there being no further procedures to invoke in response to a call, the current derivation does not lead to a refutation and must be undone (in some sense). This aspect of computation is normally implemented by backtracking, to be described shortly.

2.3.1 Search Tree

The combined behaviour engendered by the normally implemented selection and search strategies is termed left-right, depth-first (LRDF) search. The name derives from consideration of the search tree for the given goal statement and program.

The search tree for a given goal statement and program is determined by the assumed selection strategy and describes all possible ways of solving the goal statement under that strategy.

Nodes of the search tree represent goal statements (the root represents the initial goal statement). A path from the initial node to a tip node labelled by the empty clause represents a successful computation - i.e. a refutation.

A simple example of a search tree relates to the Fallible Greek problem (Figure 1).

```
Fallible(x) <- Human(x)

Human(Turing)
Human(Socrates)

Greek(Socrates)

<- Fallible(y) & Greek(y)
```

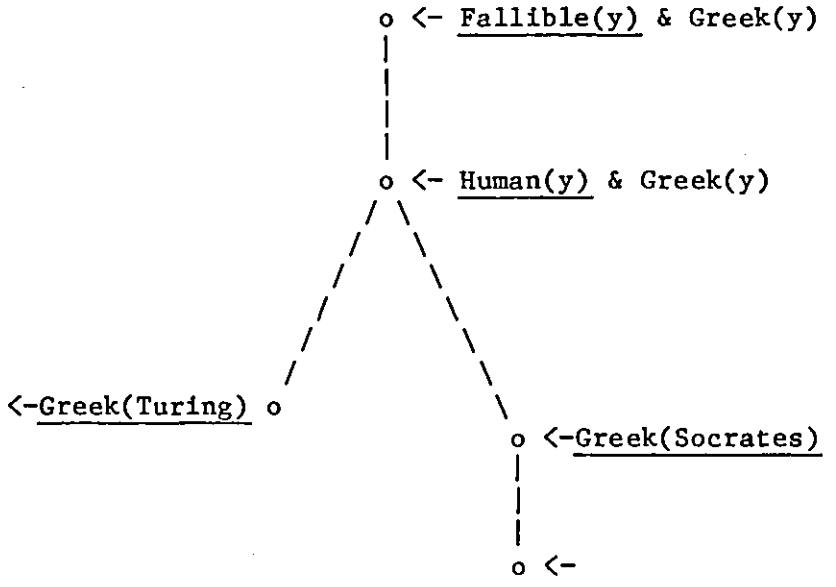


Figure 1.

The underlined atom is that chosen according to the adopted selection strategy. The search strategy normally used in Prolog implementations, as described above, will search the leftmost branch of such a tree as deeply as possible before searching the next left-

most branch. Thus Human(Turing) is tried before Human(Socrates) when solving <-Human(y). Hence the name 'left-right depth-first search' for the combined selection and search strategies.

If a derivation does not lead to a refutation, a conventional Prolog interpreter backtracks to the most recent choice point - i.e. it reinstates the computation to the state it was in at the most recent node which still has an untried child node and then selects as the next clause the clause corresponding to that child node.

2.3.2 Control

It should be apparent that the behaviour of a logic program on execution is determined by two aspects viz. the contents of the program and the way that procedure calls and definitions are chosen. Kowalski has documented this feature of (logic) programming in his paper 'Algorithm = Logic + Control' [27].

It should be stated that few implementations of Prolog offer anything but the basic control of LRDF search and of those that do, all control is manual - i.e. under the programmer's jurisdiction. Perhaps the most ambitious implementation of control features is to be found in the research version of Prolog, IC-PROLOG [6].

Anything here other than a detailed exposition would not do justice to the subject matter and because control features do not play a large part in the body of the thesis, we do not feel justified in giving such an exposition here. Instead we refer the interested reader to the above paper.

2.3.3 Negation as Failure

There is no facility in the language of Horn clauses of having a

goal statement that contains a negated atom. Such a facility is a necessary one in practice and Prolog provides it, albeit at the cost of a weaker interpretation of negation - negation as failure to prove.

In Prolog, the literal not-P in a denial is interpreted as a challenge to show P is not provable - i.e to show that all ways of solving the atomic denial <-P fail. (A literal is an atom or the negation of an atom.)

Clark has shown [4] that the failure to prove P, an observation made at the meta-level, is a semantically acceptable way of proving ¬P at the object level, provided that "if" definitions are re-expressed in "if and only if" form.

As an example of the need for negation in practical logic programming, consider the procedure set for checking that some item is not a member of a given list:-

```
Not-in(u, NIL)
Not-in(u, v.w) <- Diff(u, v) & Not-in(u, w)
```

The Diff predicate must be defined so as to hold for every distinct pair of variable-free terms in the universe of discourse. Although such a definition is possible in theory, it is not generally feasible in practice and a more convenient way of achieving the same end is by means of the procedure set

```
Not-in(u, NIL)
Not-in(u, v.w) <- not-(u=v) & Not-in(u, w)
```

and the clause

```
z=z
```

which serves to define '='. When presented with the denial <-Not-in(A, A.NIL), only the second procedure definition can be used and this results in the call not-(A=A) being made. The proof of A=A is then attempted (i.e. the main problem is held pending in favour of

the denial <-(A=A)) and succeeds with the most general unifier {z/A}. The proof of <-Not-in(A, A.NIL) therefore fails.

The call Not-in(A, B.NIL) is essentially the problem of demonstrating the failure to prove <-(A=B). The proof of <-(A=B) fails and so the original call succeeds.

Negation interpreted as failure to prove is weaker than the conventional interpretation of negation. For example, the standard form sentence

$$P \vee \neg(\neg P)$$

asserting the truth of the atom P gives rise, on 'translation' into Prolog, to the implication

$$P \leftarrow \text{not-}P.$$

Unfortunately, any attempt by a Prolog interpreter to solve <-P will result in a loop unless the interpreter has a suitable loop detection facility (loop detection facilities are not normally incorporated in practice).

A second characteristic of negation interpreted as failure is that none of the variables in the terms of the negated call are allowed to be instantiated in the course of the nested proof attempt if those variables are shared with other atoms in the goal statement.

In practice, this and the former weakness do not detract from the intuitive notion of negation. Negation as failure is consequently much used.

2.3.4 Built-in Predicates

To be of practical use, the language of Horn clauses must further be augmented by so-called built-in predicates.

For instance, the sum relationship, which holds between three integers a, b, c whenever c=a+b, cannot conveniently be expressed in the user's program. Moreover, it is unreasonable to require users to be bothered with such standard details and so all practical Prolog interpreters allow the programmer to use certain calls while the interpreter internally implements the corresponding procedure set.

Interpreters vary as to the number and extent of the built-in predicates they supply. The sum relation is an example of an arithmetic relation. Arithmetic built-in predicates are, as a rule, useful and safe additions to the language.

A second major class of built-in predicates, the meta-level predicates, are more contentious. Meta-level predicates allow the program to reason about itself.

For instance, the facility of adding and deleting clauses from a program in the course of a proof is often available. The semantics of such operations are in themselves not well-defined - for it is obvious that the order in which procedure calls are made might now be significant. Thus, the solution of a goal statement that contains a call to delete a clause in P's procedure set and also a call with predicate symbol P may well give different results depending on the order in which the calls are taken.

A second contentious class of built-in predicates are those used to control the execution of logic programs.

Consider the '/' predicate of most Prolog systems - e.g. [38]. Its declarative semantics are 'true'; its operational semantics are understood in the context of LRDF search, which it seeks to modify by excluding certain backtracking options (the details are of no consequence here).

For example the prototype IC-Prolog procedure set

```
P <- C & Q
P <- not-C & R
```

which, loosely speaking, states that the proof of P follows from the proof of Q or R, depending on whether the proof of C succeeds or fails (conventionally, P is computed by: "if C then Q else R"), is expressed in these systems as

```
P <- C & / & Q
P <- R.
```

The second clause is understood as a catch-all, the '/' predicate in the first clause overriding the use of the second clause if the '/' call is ever executed i.e. if the call C succeeds.

By removing backtracking options, '/' saves the allocation of storage and this is sometimes the reason for its inclusion.

Unlike the control features of IC-PROLOG, control implemented in this way is dangerous and can lead to complications normally associated with conventional programming languages. An example of interest in this thesis is that the use of such a feature would severely complicate the parallel invocation of alternative procedure definitions.

As a general comment, it is felt that more restraint ought to be exerted in the provision of such 'dirty' features - for often they are no more than simple expedients which serve to compensate for the deficiencies of current machines and have the side-effect of blurring the program's meaning. Considerable research effort, e.g. [1], is aimed at the 'harmless' incorporation of pragmatically sufficient meta-level facilities.

CHAPTER 3: A BASIC HORN CLAUSE INTERPRETER

3.1 INTRODUCTION

This chapter is concerned with the design of a hypothetical basic Horn clause interpreter and the manner by which it might be implemented on conventional von Neumann architecture.

One reason for including such a description is to provide a frame of reference in which comparisons between various approaches to program execution can be made. The second reason is that certain aspects of the proposed designs bear close resemblances to their conventional counterpart and so the description of a sequential interpreter will assist in their explanation.

The interpreter's specification will be given principally in Prolog although for the sake of brevity, we will content ourselves with a narrative description of certain lower-level components.

The reader interested in the details of a conventional implementation of Prolog is referred to [38], [43].

3.2 OUTLINE REQUIREMENTS OF THE INTERPRETER

The interpreter will be presented with the user's Horn clause program and the denial which constitutes the goal statement. It will perform a left-right depth-first search in its attempt to find a refutation.

A structure-sharing [2] implementation will be described because it is typical of conventional interpreters and the proposed implementations of our schemes for parallel execution intend to make use of structure-sharing in one form or another.

3.3 ABSTRACT INTERPRETER

Before describing the mechanics of the interpreter, it is convenient here to re-state certain aspects of LRDF search which need to be borne in mind in what follows.

3.3.1 Search Strategy

Under LRDF search, the interpreter will traverse the search tree from left to right i.e. at every choice point in the search tree, the leftmost branch will initially be selected and in the event of the chosen branch not leading to a refutation, the interpreter will back-

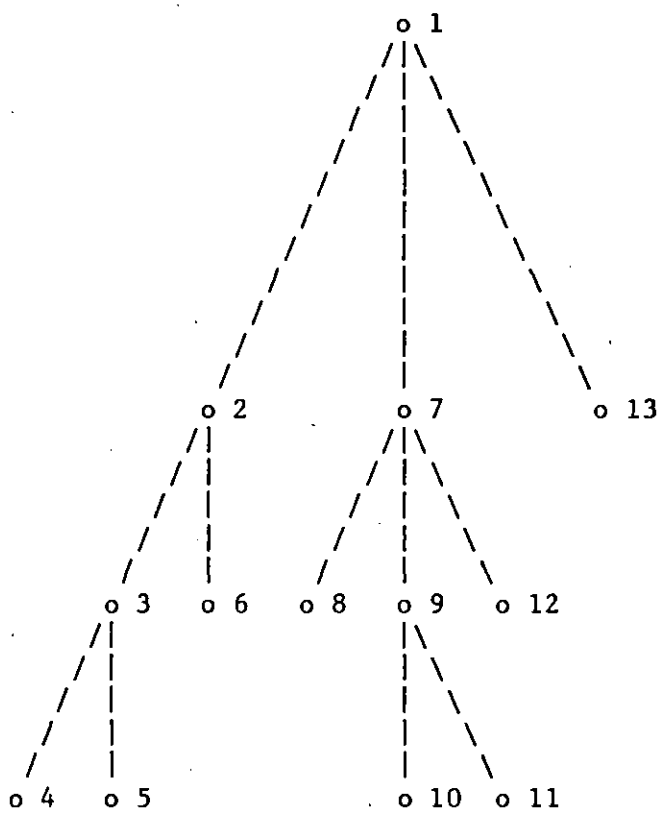


Figure 2.

track to its nearest choice point and then try the next leftmost branch, etc.. The strategy is depicted in Figure 2, where node labels reflect the order in which new branches of the search tree are explored. In this way the entire search tree is explored if necessary (assuming no infinite branch is followed).

It can be seen that backtracking implies the need to nullify (in some sense) the effects of all computations between the most recent node and the nearest remaining choice point whenever it is recognised that the current branch does not lead to a refutation. Data structures must therefore be designed to allow for this eventuality.

Because just one branch of the search tree is searched at a time, it is sufficient to represent the tree as the currently active branch and incorporate in the representation sufficient information to allow for the activation of alternative branches at some later time, if necessary. The currently active branch is normally represented as a stack of activation records, a record corresponding to each node in the branch. This aspect of the implementation will be described more fully later.

3.3.2 Selection Strategy

The selection strategy associated with LRDF search dictates that goals are selected in left-right, last-in-first-out order.

The list of goals is initially given. Whenever an implication is used in the course of refutation, the goals arising from the antecedent of the implication are added in a way which ensures that they are selected before the other outstanding goals. Within themselves, the introduced goals are selected according to the order in which the corresponding atoms appear in the antecedent of the implication used.

3.3.3 Variable Naming

It is essential that variable names are chosen in such a way that no matter how often a clause it used in the course of a proof, no confusion can arise between different instances of its variables.

Each activation record is given a unique name. By incorporating this name in the names of those variables introduced by the clause referred to in the activation record, all variable names are guaranteed unique. Moreover, in this way, variables may conveniently be associated with activation records.

Variable names take the form

<level, static variable>

where 'level' is the name of the relevant activation record and 'static variable' names the variable as it appears in the clause used.

Terms are usually represented in an analogous manner. Thus a term introduced through the invocation of a clause is represented by the ordered pair

<level, static term>

where 'level' is again the name of the activation record corresponding to the invocation of the clause which contains the term and 'static term' is a data structure which describes the term as supplied in that clause.

Interpretation of the term is carried out implicitly by reading each static variable v occurring in 'static term' as the variable

<level, v>.

3.3.4 Structure-sharing

It should be clear that the normally adopted variable naming scheme not only guarantees unique names but also makes possible the sharing of static term structures. The scheme may be extended to other expressions, not just variables and terms.

For instance, the antecedent atoms of a clause may be referred to by the ordered pair

<level, static antecedent>.

This device is used in the representation of goal lists (see below) which, of course, originate from the antecedents of clauses. The variables in each goal are named by the usual pair, the level component being that of the antecedent.

In general, such **structure-sharing** obviates the need to copy data structures and so significant economies may be forthcoming - at least in a conventional implementation - from what is in any case an elegant feature.

The conventional representation of a goal list also involves a further, more specialised, form of structure-sharing. One must bear in mind that at each non-terminal node, all but one goal (the selected goal) appears in the goal list of the immediately descended node. Copying outstanding goals is clearly wasteful and the usual approach is to share, at each node, references to outstanding goals. We will describe precisely how such sharing may come about once we have fixed the representation of node structures.

3.3.5 Bindings

When a variable is bound to a term, that term is not constructed

but is held implicitly. In other words, a binding may be viewed as the data structure

<variable, <level, static term>>.

A later unification may need to evaluate the term part of this binding and to be efficient, it is important that the evaluation is quickly able to determine whether any variable in the term is bound and if so, to what term. Such a variable will have the same level as that of the term. Rapid access to a variable's binding is achieved by the following means:-

When an activation record is established, binding space is reserved for each variable introduced by the clause used in the activation. These variables will be partly named by the level of their introductory activation record and partly by the static variable, normally represented internally as a natural number.

The determination of a binding value for a variable then reduces to using the two components of the variable's name for two direct accesses. The first access locates the activation record and the second interprets the static variable name as a displacement within the activation record.

In the event of backtracking, one or more activation records will need to be deleted. Such deletions will automatically remove the bindings made to variables introduced by the activation records concerned. These bindings clearly cannot have been made by earlier unifications nor by later ones since they would have been undone by earlier backtracking. Therefore the bindings deleted with an activation record are all part of the unification being nullified - in fact, they constitute the input portion of the unifier. The output portion will be distributed throughout earlier activation records and the variable in each output component has to be reset to its previously unbound state.

The means by which this is normally achieved is to associate with each unification a reset list naming all earlier variables that were

used to transmit output from the unification. Then in the event of having to undo a unification (or indeed a partial unification resulting from a failure to unify), all that is required is to use the reset list to access and unbind the named variables.

It will be appreciated that whenever a variable-variable binding is to be made, if one of the variables is new (i.e. just introduced into the computation), it is best to make the other variable the term component of the binding and thereby place the binding within the most recent activation record. One reason for doing this is that no entry in the reset list is then required. Another is that doing so tends to produce shorter variable-variable chains of bindings. (Further reasons are concerned with certain optimisations which are of no interest here.)

3.4 IMPLEMENTATION

The specification we give here is at a lower level than others (e.g. in [28]) because our concern is to give a clear indication of how a basic Horn Clause interpreter might be implemented on a traditional von Neumann machine.

We choose an implementation based on the state of such a machine, which we represent as a list (stack) of activation records.

The top-level of the program defines the 'Demonstrate' relation between the state of the machine and the supplied Horn clause program.

The meta-level goal statement takes the form

```
<-Demonstrate(program, initial-state)
```

where 'initial-state' is a singleton list whose activation record describes the supplied goal statement in one of its arguments. Notice that this relation will not return a solution. In order to do so, Demonstrate would need to be a three place predicate, the third argu-

ment representing the final state (see later). We describe the simpler formulation in the interests of clarity.

Each activation record is represented by a term of the form

ar(level, goal, clause, bindings, reset)

where

'level' is the (unique) name of the activation record

'goal' names the goal which the activation sets out to solve. In a conventional implementation this argument is a pair

<level', goal number>

where the first argument names the activation record that introduced the goal. The third argument of this earlier activation record names the clause which was used at level'. The antecedent of that clause gave rise to subgoals, one of which is the goal in the more recent activation record. Exactly which of those subgoals is specified in the later activation record is indicated by the 'goal number' component of its goal argument. We will shortly make precise how this structure implements the goal list at any node.

'clause' names the clause being tried in order to solve the goal. In practice, it would be a reference pointer to a clause in the program.

'bindings' is a data structure that indicates, for each variable introduced into the computation by the present activation, whether the variable is bound and if so, to what. In practice it would be an array, one entry for each variable introduced by the clause.

'reset' is the reset list, containing an entry for each earlier variable bound in the course of unification performed in the current activation.

3.4.1 Goal List

Before embarking on the detailed description of the backtracking interpreter, we wish to make clear exactly how a goal list is represented. We will illustrate the description with the simple stack of activation records outlined in Figure 3. (For the sake of clarity, the clause argument of an activation record is not represented here as a reference pointer to the clause; instead, just the antecedent of the clause appears, the atoms within it being represented by numbers. Bindings and reset lists are not illustrated. The stack is shown grown upwards with the most recent record on top.)

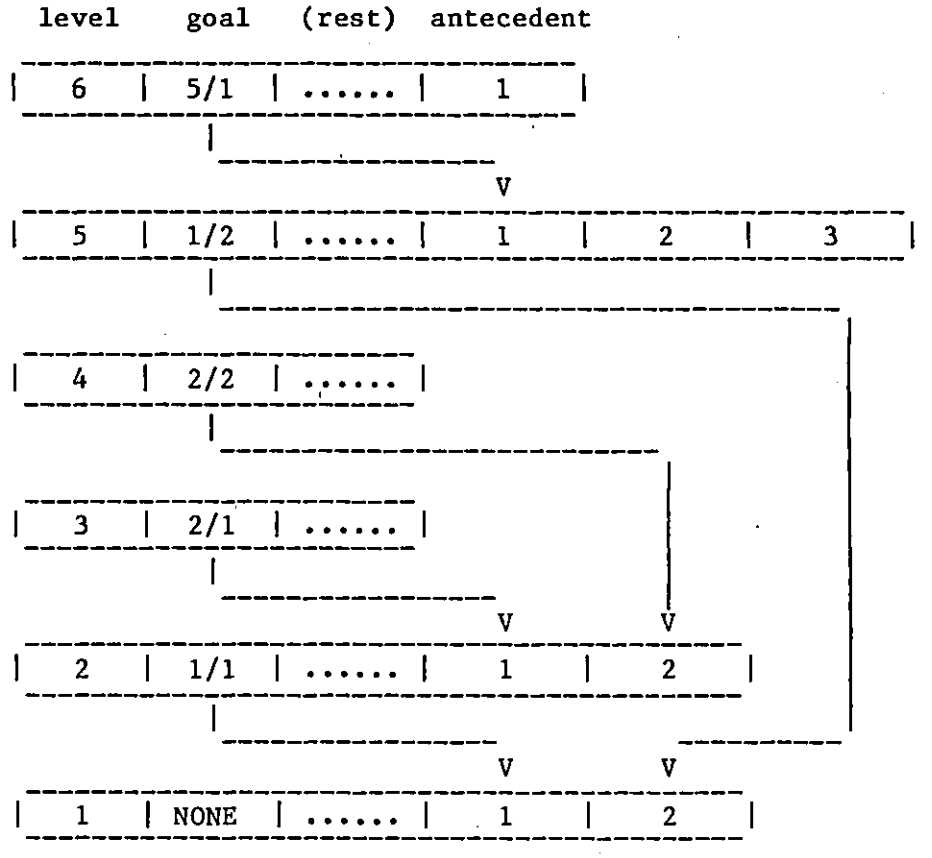


Figure 3.

The diagram contains six activation records whose antecedents introduce respectively 2, 2, 0, 0, 3, 1 subgoals.

The goal argument is shown as a pointer to the selected subgoal in some earlier activation record and we now make clear how the left-right, last-in-first-out selection strategy is implemented.

Essentially, if the last used clause was an implication, the goal selected for the next level arises from the first atom in the antecedent. Thus, for example, the clause used at level 5 was an implication and so the goal selected for level 6 was 5/1.

If the last clause was an assertion, the next goal (if any) is found by examining in turn the list of activation records determined by the chain of previously selected goals, until an activation record with an outstanding subgoal is discovered. The first such subgoal then becomes the goal for the next activation record. For example, the clause used at level 4 is an assertion and activation record 2 introduced that level's goal. It can be seen that no further goals introduced at level 2 remain and so the level 2 goal, viz. 1/1, leads to the activation record at level 1 being examined and this shows that 1/2 is the next goal to be selected - which therefore becomes the level 5 goal.

3.4.2 Description of Processing

See the appendix.

CHAPTER 4: PARALLELISM

4.1 INTRODUCTION

The Resolution Theorem [39] is the general result upon which the execution of Horn clause logic programs is based. Computation, as stated in the Resolution Theorem, is impractical for all but the most simple examples. The work of Loveland on Model Elimination [30], re-discovered by Kowalski and Kuehner [29] and applied to programming by Kowalski [25] and Colmerauer et al. [10], have made resolution a viable computational mechanism for Horn clauses and certain extensions.

A computational step in the procedural interpretation selects and activates a single procedure call. A conventional implementation of this interpretation imposes the added restriction of only invoking a single definition in response. Jointly, the procedural interpretation and its conventional implementation serve to suppress all the parallelism inherent in Horn clause programs. This is hardly surprising in view of the relationship that (such an implementation of) the interpretation bears to the classical notion of computation which, of course, makes no allowance for parallel computation either. What is needed is a compromise between the resolution theorem and conventional implementations that is computationally viable and yet allows for parallelism.

We now outline the various forms of parallelism which might be exploited in the execution of Horn clause programs.

4.2 POTENTIAL FOR PARALLELISM

The procedural interpretation places no constraint on independent derivations being pursued in parallel. Consequently, one way of

applying parallel computation to Horn clause programs is to allow different derivations to take place concurrently.

Because of the completeness result for top-down inference systems proved by Hill [20] and others, there is no loss in generality in specifying the same selection function for all such derivations. For a given Horn clause program, goal statement and selection strategy, there exists a well-defined search tree. Parallel exploration of this tree's branches is the pursuit of parallel derivations and since the branches arise from alternative ways to solve the selected goal, this form of parallelism is termed **Or-parallelism** and the search strategy parallel search.

The procedural interpretation does not commit itself to sequential invocation of procedure definitions and so we may say that the procedural interpretation will support or-parallelism if all procedure definitions whose head potentially matches the selected subgoal are 'simultaneously' invoked in response to it. Separate derivations will thereby be established.

The second principal way of exploiting parallelism in Horn clause programs arises from consideration of resolution itself. Suppose the goal statement

$$\leftarrow P_1 \ \& \ P_2 \ \& \ \ldots \ \& \ P_n,$$

is given and that call P_i is selected. After one resolution step, an instance of all the other calls P_j will remain outstanding, the instance being determined by the output component of the substitution which results from the unification of P_i with the head of the chosen procedure.

As a special case, it might happen that the particular instance computed for the call P_j (say) is the identical instance. Such an eventuality might arise if P_i and P_j shared no variables or resolution of P_i binds none of the variables P_i shares with P_j. In these events, P_j could have been selected simultaneously and concurrently subjected to a step of resolution, suitable steps being taken to compose both substitutions.

A general scheme for accommodating such special cases is to allow solutions of the calls P_1, P_2, ..., P_n to proceed in parallel, recognising that each of the calls P_1, P_2, ..., P_n being executed may represent an instance of the respective goal (the instance determined by the output components of substitutions produced in the execution of other calls). Suitable communication has then to be provided in order that the instances may be deduced. We will view this informal description as the relaxation of the procedural interpretation needed in order to support the multiple selection of procedure calls.

Because the calls in a goal statement are connected by AND (&) operators, this form of parallelism is termed **And-parallelism.**

A third area in which parallel computation might be allowed is in the unification algorithm itself. Obviously, this is parallelism at a lower level and is applicable to any scheme, since all schemes use unification. We will not therefore concern ourselves with it in this chapter, where the principal aim is to describe the application potential of higher level parallelism. However, one of our schemes, the And-or proof procedure, is designed in such a way that a parallel unification algorithm fits in very naturally and we will take the opportunity of describing concurrent unification there. The other scheme, the Or-parallel proof procedure, also has a certain degree of lower level parallelism.

Having outlined the ways in which parallel execution might be adopted, a motivating description of how such parallelism might be used to good effect is overdue and we now turn to such a description for the two principal areas, Or- and And- parallelism.

4.3 OR-PARALLELISM

4.3.1 Database Applications

It is a default of many Horn clause interpreters that they stop

when just a single solution of the user's goal is found. It might be argued that for many applications expressed in a relational language, a more natural requirement is to compute the relation that solves the user's goal - not a single member of it - and this is the default we adopt here. The view is in concordance with database applications, arguably the most natural for the adoption of or-parallelism, where the user is normally interested in all solutions of his query, not just the first one discovered. Such applications can be expected to benefit from the adoption of or-parallelism.

4.3.2 Functional Problems

If the user's program and goal admit just a single solution, it might well be asked what benefits accrue from the adoption of Or-parallelism. Certainly, such a combination is common in practice and so this question has some importance.

The nub of the answer is that even if there is only one solution of the top-level goal, the same need not necessarily be true of any individual subgoal. It may be that some branches of the search tree need to be explored to a non-trivial depth before a failure is found and if the search tree is investigated sequentially, as in a backtracking interpreter, this process may take a significant length of time, time which does not contribute to the discovery of the solution. If the searches could be performed concurrently, then (ideally) no time would be wasted in following dead-end branches, only computing effort. Moreover, if the used computing power would not otherwise have been employed then no real loss is entailed by this 'waste' of effort.

The naive sort, which sorts an input list by generating permutations of it and testing them for orderedness is, under LRDF control, an example to hand.

```
Sort(x, y) <- Perm(x, y) & Ord(y)

Perm(NIL, NIL)
Perm(u, v.w) <- Delete(v, u, w') & Perm(w', w)

Delete(u, u.x, x)
Delete(u, v.x, v.y) <- Delete(u, x, y)

Ord(NIL)
Ord(u.NIL)
Ord(u.v.w) <- u ≤ v & Ord(v.w)
```

Although useless as a practical program, it does illustrate that considerable effort might be expended in following fruitless paths, in this case, by generating complete permutations.

Parallel search is fairer than depth-first search in that not all computing resources are committed to the exploration of a single branch of the search tree. Depth-first search proves disastrous whenever the chosen branch is infinite (assuming the condition is not detected).

Breadth-first search, in which computing resources are switched so that all nodes at level N in the search tree are investigated before any of those at level N+1, shares the property of fairness with parallel search. However, it is not normally implemented because firstly it 'dilutes' the power available from a single processor by applying it to the exploration of all branches (it is therefore, in general, slower to find the first solution) and secondly the act of switching from branch to branch is, in general, a significant overhead. An implementation of Loglisp (Logic in Lisp), essentially a breadth-first implementation of Horn clause logic, is described in [40]. As one might expect, the implementations of parallel and breadth-first search share much in common. With the availability of more computing power, or-parallelism might reasonably be expected to overcome the implementation difficulties of breadth-first search.

An alternative approach to parallel investigation of the search tree is the $\top\top$-Representation proposed by Fishman and Minker [17].

Briefly, they achieve parallel search through their choice of clause representation. In this representation, a set of syntactically similar clauses is represented by a single $\overline{\Pi}$-clause and the notions of unification and resolution are extended accordingly. The overall effect is that syntactically similar derivations are pursued in parallel although the corresponding search tree need only be investigated sequentially. Their proposal is primarily aimed at practical database applications where large ground relations, which might otherwise prove difficult to search efficiently, are quite common.

4.3.3 Negative Literals

Although this thesis is primarily concerned with Horn clause programming, the negation as failure inference rule is necessary in practice and its implementation in an or-parallel environment calls for some comment here.

The rule is 'infer ¬P if all ways to prove P fail', where P is an atom and no attempt is made in the nested proof attempt to instantiate any variable that P shares with other goals.

The rule implies that an exhaustive search of the nested proof's search tree should be made and, as in the top-level proof, a parallel search seems natural (although the search may be abandoned if a solution which does not bind any shared variable is found).

4.3.4 Implicative Literals

Our interest here is in procedure calls of the form

```
all(Q -> R)
```

where Q and R represent atoms, some of whose variables may be universally quantified in the procedure call ('all' representing those quantifiers).

We will term such a call an 'implicative literal' (and ignore the more general case where Q and R represent conjunctions of literals - which in any case may be transformed into the above form). For example,

```
Subset(x, y) <- All z ( Member(z, x) -> Member(z, y) )
```

"x is a subset of y if for all z, z is a member of x implies that z is a member of y".

This construct occurs quite naturally in specifications and one would like to execute it directly rather than transforming it into clausal form through Skolemization [28].

We briefly show how this construct may be transformed into Horn clauses augmented with negation and describe the construct's semantics and the operational behaviour of its execution if negation is interpreted as failure to prove.

Introduce the definition

```
P <-> ¬all(Q -> R)
```

where P's predicate symbol is not used elsewhere in the program and the terms of P are all variables, namely those variables in the formula which are not quantified by 'all'.

The definition allows the implicative literal to be replaced in the clause where it occurs by the negative literal ¬P. Since P's predicate symbol is not used elsewhere in the program, the only way to solve P is by means of

```
P <- ¬all(Q -> R)
```

which is readily transformed into the clause

P <- Q & ¬R,

all quantification being universal outside the clause.

Suppose now the literal ¬P is selected. For the proof of ¬P to succeed, all ways to solve P must be tried and shown to fail. The only way to solve P is via the above clause for P. So, the task of showing all attempts to prove P fail is equivalent to that of showing that all attempts to prove Q & ¬R fail.

The only way in which a proof of Q & ¬R could possibly succeed is if a solution of Q gives rise to an instance of R whose proof fails.

Equivalently, to show that all proofs of Q & ¬R fail, it is necessary to show that for each solution of Q, the instance of R determined by that solution itself has a solution.

Thus to show <- Subset(1.3.NIL, 1.2.3.4.NIL), the rule stipulates that all solutions of <- Member(z, 1.3.NIL) must be found and then used to instantiate the goal <- Member(z, 1.2.3.4.NIL) and that each such goal admits a solution.

The 'no instantiation' rule of negation as failure requires that none of the shared variables in P are allowed to be instantiated in the nested proof attempt. Since the arguments of the P-goal and P-head are variants of one another (identical save for an identity mapping of variable names), their unification need not instantiate any goal variable. Therefore, the restriction that none of the shared variables in the P-goal be instantiated may equivalently be re-stated as the restriction that none of the variables in the goal <- Q & ¬R be instantiated unless they are local to Q & ¬R (i.e. those variables originally quantified by 'all').

We may summarise the above in a way which removes all reference to the intermediate definition of P.

The inference rule for solving goals of the form <-all(Q -> R) is

1. Find all substitutions, S, arising from the solution of the goal <-Q

2. For each such S, prove <-(R)S

3. The proofs of <-Q and <-(R)S are only allowed to instantiate variables quantified by 'all'.

Or-parallelism is appropriate for the direct implementation of the implicative inference rule since the rule requires all solutions of <-Q to be found.

4.4 AND-PARALLELISM

4.4.1 Independent Subgoals

Possibly the most obvious way to envisage the need for and-parallelism is in computations where some subgoals are independent in the sense that they share no variables. Certainly, it is most inappropriate to use a sequential proof procedure in such cases as sequential execution gives rise to poor behaviour on two counts. We illustrate these deficiencies with the help of the example <-P(x) & Q(y) executing under LRDF control, where all solutions are to be found.

Firstly, and most obviously, solution of the goal <-Q(y) could proceed independently and concurrently with that of <-P(x), thereby speeding up overall computation, assuming appropriate resources are available.

Secondly, and in general more importantly, a backtracking interpreter will find a single solution of <-P(x) and then apply the substitution which represents that solution to <-Q(y), and solve the instance of <-Q(y) thus derived. When the goals share no variables,

the derived instance of <-Q(y) is, of course, the identical instance. The backtracking interpreter then finds the second solution of <-P(x) and follows this with an exactly repeated computation of <-Q(y). If the number of solutions of <-P(x) and <-Q(y) are m and n respectively, then the backtracking interpreter will expend effort of the order m*n whereas a parallel interpreter solving those subgoals independently will expend effort of the order m+n.

We give a simple example of how disastrous the backtracking strategy can be.

```
P(NIL, NIL)
P(u.x, v.y) <- Q(u, v) & P(x, y)

Q(u, v) <- v = 2*u
Q(u, v) <- v = 3*u
```

The example accepts an input list of integers of length n (the first argument of P) and computes a set of output lists, all of length n, wherein each item is either two or three times the magnitude of the corresponding item of the input list.

It is readily verified that the parallel computation does O(n) units of work whereas the backtracking computation does $O(2^n)$ units of work.

4.4.2 Pipelining

Another way in which parallelism might be used is in 'pipelining mode'. Typical of such applications is the production and consumption of lists, whose items flow one by one along the pipelines. Pipelined parallelism is specifically addressed by Clark and Gregory in [5]. Below we give one of their examples in a simplified form.

```
Compact(NIL, NIL)
Compact(u.x, u.y) <- Remove(u, x, z) & Compact(z, y)

Remove(u, NIL, NIL)
Remove(u, u.w, w´) <- Remove(u, w, w´)
Remove(u, v.w, v.w´) <- ¬(u = v) & Remove(u, w, w´)

x = x
```

Lists u and v are in the Compact relation if list v is the same as list u except that duplicates of earlier list u items are not present in list v. For example Compact(3.2.3.1.1.NIL, 3.2.1.NIL) holds.

We intend that the first list is given and the second is to be computed.

Suppose that the program is presented with a goal of the form

```
<- Compact(3.2.3.1.....NIL, t).
```

The program for Compact causes the first item of the output list to be bound to the first item in the input list and calls Remove to create an intermediate list identical to the tail of the input list except all occurrences of the first item are missing from it. It also calls Compact to form the tail of the output list from the intermediate list. The essential behaviour of the recursive Compact clause is depicted in Figure 4 on page 54, from which it should be readily appreciated that expansion of the nested Compact box will result in a string, or pipeline, of Remove computations.

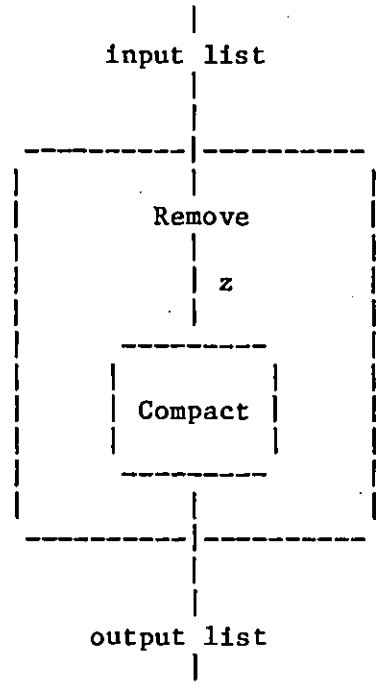


Figure 4.

Suppose that the Remove(3, 2.3.1..., z) call has been selected. Only the head of the third Remove clause will unify with this goal and once this has been done, the first item in list z will be known. The outstanding call, Compact(z, y) now has some input - it knows the first item (2) of list z and so can compute the second item of list t. Moreover, as further items of list z are computed, solution of the call Remove(2, 1..., z´) can proceed and thereby the third item in list t may be computed and used to establish a new filtering Remove process etc..

The pipeline shown in Figure 5 on page 55 will have been established.

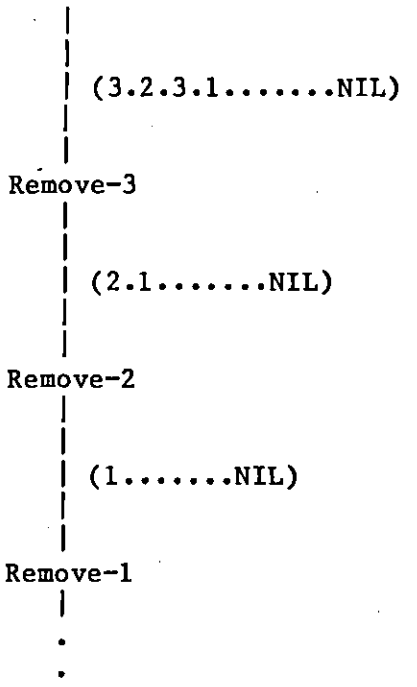


Figure 5.

The pipeline can be thought of as connecting the various Remove-n computations in a manner that allows each to still be executing whilst its later neighbour is busy filtering its own input, essentially the partially known output of its predecessor.

We briefly mention a second and perhaps more practical example of this view of computation. The example is the top-level of Hoare's Quicksort [21], given below.

```
QS(NIL, NIL)
QS(item.in, out) <- Partition(item, in, low, high) &
                    QS(low, s-low) &
                    QS(high, s-high) &
                    Interpose(s-low, item, s-high, out)
```

Assuming the first argument in the QS call is ground and the second is a variable, the pipeline behaviour of the recursive QS clause is summarised in Figure 6 where arcs denote the communication of list structures.

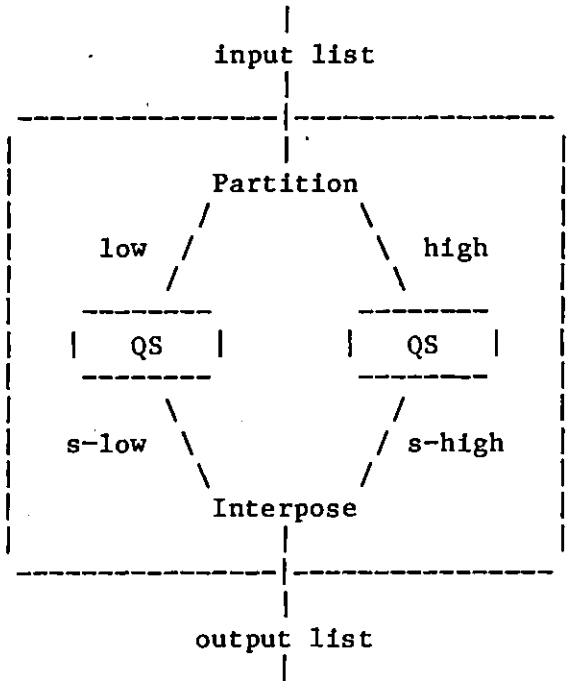


Figure 6.

If now the nested QS boxes are expanded, the structure results in a more complex pipeline (more accurately, network), of the form depicted in Figure 7 on page 57.

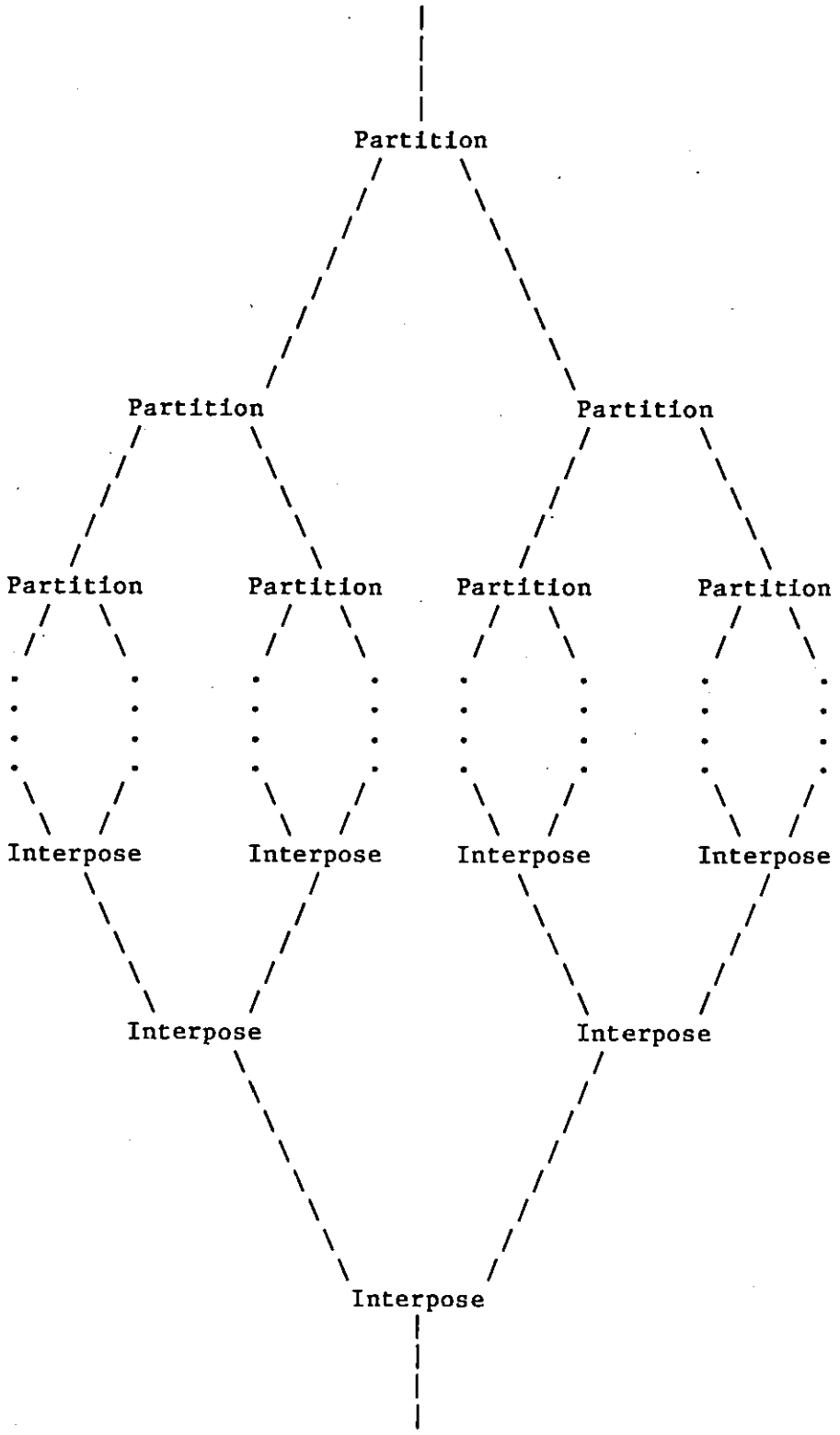

Partition
Partition
Partition
Partition
Partition
Partition
Partition
Interpose
Interpose
Interpose
Interpose
Interpose
Interpose
Interpose

Figure 7.

4.4.3 Early Detection of Failure

Another way in which and-parallelism may be exploited is in the early detection of failure. This aspect is exemplified by the Same-leaves program which seeks to show that the leaflists of two binary trees are the same. Figure 8 illustrates two trees with the same leaflist (B.D.A.C.NIL).

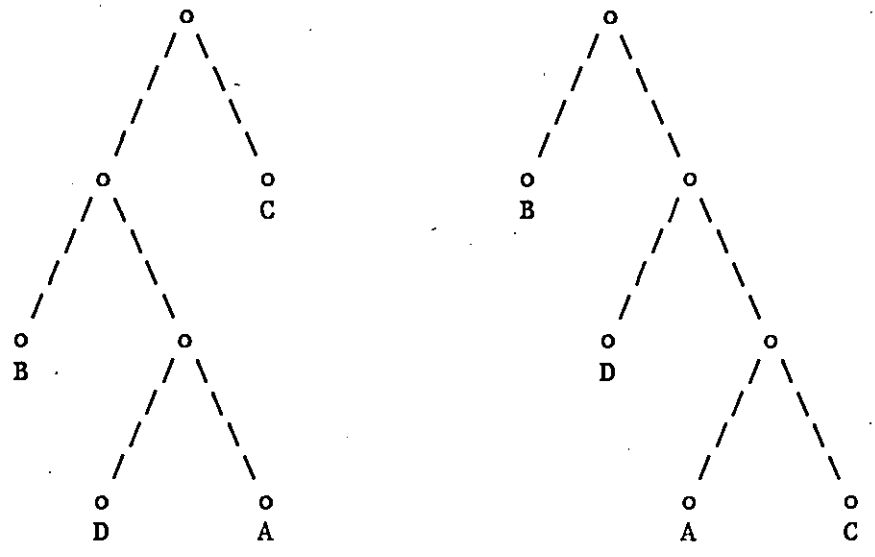


Figure 8.

The program for Same-leaves is given below.

```
Same-leaves(tree1, tree2) <- Leaflist(tree1, list) &
                             Leaflist(tree2, list)

Leaflist( l(leaf), leaf.NIL)
Leaflist( left:right, list) <- Leaflist(left, leftlist) &
                               Leaflist(right, rightlist) &
                               Append(leftlist, rightlist, list)

Append(NIL, x, x)
Append(u.x, y, u.z) <- Append(x, y, z)
```

Here, a tree is either a single leaf tree - represented by 1(x) where x is the leafname - or a compound tree - represented in infix form by the term left:right where 'left' and 'right' name the left and right subtrees respectively. The terms representing the example trees in Figure 8 on the previous page are

(1(B) : (1(D) : 1(A))) : 1(C) and
1(B) : (1(D) : (1(A) : 1(C)))) respectively.

Suppose now that this program is presented with a goal statement of the form

<-Same-leaves($tree_1$, $tree_2$)

where '$tree_1$' and '$tree_2$' represent fully instantiated trees.

If the program is executed using the last-in-first-out selection rule then the Leaflist($tree_1$, list) call and all calls introduced by it will be fully executed - and 'list' will be fully instantiated - before the Leaflist($tree_2$, list) call is selected. The consequence of this is that considerable effort may have been devoted to the entire evaluation of $tree_1$'s leaflist even though only a short investigation into $tree_2$'s leaflist may have been sufficient to determine that the two trees have dissimilar leaflists.

The effect is exaggerated in the extreme case where $tree_1$ and $tree_2$ have dissimilar single leaves - A and B respectively - on their left branches and arbitrarily complex tree structures on their right. If evaluation of the trees' leaflists were able to proceed in parallel, relatively little computation would be needed to establish that the output leaflist, named 'list', would simultaneously need to satisfy the two subgoals

<- Append(A.NIL, $r\text{-}list_1$, list)
and
<- Append(B.NIL, $r\text{-}list_2$, list)

(where $r\text{-}list_1$ and $r\text{-}list_2$ are respectively the leaflists of the right subtrees of $tree_1$ and $tree_2$). If now one of these Append calls is

scheduled - say the first - then 'list' will be bound to a term of the form A.list' and the other Append goal,

$$\text{<- Append(B.NIL, r-list}_2\text{, A.list')},$$

will fail and thereby cause the top-level Same-leaves goal to fail.

Ideally, the evaluation of the right subtree leaflists can proceed independently of these two Append subgoals. In the event of the failure we described above, their evaluation can be aborted. In the absence of such failure (modifying the example so that the two leftmost leaves are the same) their evaluation can be pursued in parallel, subject to similar communication through the shared variable list' (which will have been bound to both r-list_1 and r-list_2 by the successful solution of the two Append subgoals).

(Although somewhat out of place here, it is worth pointing out the power of unification which this example illustrates. The top-level of the program,

```
Same-leaves(tree1, tree2)
  <- Leaflist(tree1, list) &
     Leaflist(tree2, list)
```

hands over to unification the task of showing that the two leaflists are identical. A lower level version of this program, one which is list-structure dependent, might be

```
Same-leaves(tree1, tree2)
  <- Leaflist(tree1, list1) &
     Leaflist(tree2, list2) &
     Same-items(list1, list2)

Same-items(NIL, NIL)
Same-items(u.x, v.y)
  <- u = v &
     Same-items(x, y)

z = z
```

It will be appreciated that such power can be used to good effect by raising the level at which the user writes his program.)

4.5 REGULATION OF PARALLELISM

The potential for parallel execution of Horn clause programs, as described in the preceding section, is only constrained by the nature of the problem being solved.

For example, the use of an or-parallel scheme on a deterministic program results in essentially sequential execution.

On the other hand (and of more interest here), parallelism applied to other problems, for example a naive formulation of the Eight Queens problem [23], might result in a great deal of parallelism.

It is generally accepted that too much parallelism may be more harmful than too little - for there is then the risk of overloading the underlying execution mechanism, possibly precipitating a failure. Hence, some means of controlling the allowable degree of parallelism in any scheme must be devised.

The need for control can be seen from the following intuitive argument (to which no claims of rigour are attached).

Performing a computation in parallel entails the sharing of resources - since the machine's resources need to be allocated to 'sub-computations'. As with all forms of resource sharing, whether in a computing context or more generally, communication is necessary to ensure proper sharing - e.g. to ensure that two users do not attempt to use the same resource at the same time or that resources no longer required are made available to others.

The penalty for such sharing is a communications overhead. Provided the benefits of sharing outweigh the communications cost, resource sharing is worthwhile.

In an ideal world where no such communication penalty is incurred, parallelism, by overlapping sub-computations, may be expected to increase the speed of computation. Introduction of the mandatory communication penalty, however, increases the cost of computation - because computational power is needed to handle the necessary communication.

If insufficient computational power is available, queues of outstanding work will build up, machine overloading will occur and the speed of computation will decrease, thus detracting from the benefit of adopting parallel execution. In this way, it might well happen that the adoption of parallel execution is entirely counter-productive.

We see the rudiments of performance as being described by a four-place relation:-

Performance(speed, available-power, required-power, parallelism)

Normally, 'available-power' is fixed because the particular machine is fixed. Provided the degree of parallelism is such that 'required-power' does not exceed 'available-power', 'speed' might be expected to increase with increasing 'parallelism'. Once the degree of parallelism causes 'required-power' to exceed 'available-power', 'speed' decreases with increasing parallelism.

Maximum speed is obtained when the degree of parallelism causes the required power to exactly match available power.

To conclude this section, we will briefly discuss the factors which influence 'degree of parallelism' as used above.

Following Kowalski [27], we consider an algorithm (A) to depend on two components, Logic (L) and Control (C), symbolically, A = L+C. In this analysis, we consider control to be determined solely by the nature of the proof procedure used.

Execution of the algorithm determines the degree of parallelism referred to in the Performance relation above. Thus the degree of

parallelism may be viewed as partially derived from the logic component of the algorithm and partly from the proof procedure used to execute it.

In support of this conjecture, consider two programs, P_a and P_o. In P_a, all procedures are deterministic - i.e. at most one definition can be used in response to a given call. In P_o, this restriction does not hold but instead, each definition has at most one call in its antecedent and the goal statement consists of just one call. The programs P_a and P_o together with their goal statements establish logic components L_a and L_o.

A proof procedure, C_a, which supports and-parallelism but not or-parallelism will, in general, give rise to more parallelism in the algorithm C_a+L_a than in the algorithm C_a+L_o. A corresponding argument holds for the proof procedure C_o, which supports or-parallelism but not and-parallelism.

Thus it is evident that for good performance (if the above intuitive arguments are accepted) a means of controlling the degree of parallelism must be provided and the control mechanism should be activated in response to perceived machine behaviour. Dynamic monitoring of machine performance is therefore needed and, of course, some allowance must be made for the resources needed to do such monitoring.

4.6 INTRODUCTION TO THE SCHEMES

This chapter concludes the first part of the thesis. The second part will be concerned with two schemes, the Or-parallel proof procedure and the And-or proof procedure.

As its name implies, the first of these schemes only exploits or-parallelism, i.e. it achieves concurrency by parallel pursuit of alternative derivations. It is put forward as a short-medium term proposal which is realisable now through existing technology.

The second scheme, again as its name implies, exploits both of the major forms of parallelism inherent in Horn clause programs. Its organisation is quite different from that of the first scheme and is developed indepedently of it - although there is a relationship between the two schemes which we exhibit at the appropriate point of the exposition. The And-or proposal is put forward for implementation in the longer term.

We have no proposal which allows for just and-parallelism. We consider this to be covered by the work of Clark and Gregory [5] and view their scheme, albeit for a different language, as a complement of ours.

CHAPTER 5: OR-PARALLEL PROOF PROCEDURE

5.1 INTRODUCTION

The relationship between the backtracking and Or-parallel proof procedures is close enough to allow us to present the latter as an evolution of the former, described previously. Indeed, for the most part, this chapter is concerned with possible implementation designs; the abstract requirements of the proof procedure are summarised in the following paragraph.

5.2 BASIC REQUIREMENT

The fundamental requirement of the Or-parallel proof procedure is that given a selection function, branches of the corresponding search tree are to be explored in parallel. That said, the remainder of this chapter is almost exclusively concerned with how this requirement may be implemented in practice.

5.3 IMPLEMENTATION DECISIONS

The principal implementation decision to be made is to settle the question of whether a structure-sharing approach is appropriate or not.

Our belief is that a structure-shared implementation is desirable for the execution of general purpose logic programs; the reasons underlying this belief are essentially the same as those applying to conventional implementations. In those implementations, structure-sharing is usually applied in two principal ways (as dis-

cussed previously) - in the representation of bindings and in the representation of goal lists. Were structure-sharing to be rejected in both cases, one would need to construct terms and explicitly apply substitutions to all outstanding goals after unification, copying those goals at each branching node.

Our belief is that the processing required to implement the above approach would typically be too expensive for practical application. It is really quite common to find long and complex terms being constructed in the course of a derivation.

Consider, for example, the simple Append procedure set

Append(NIL, x, x)
Append(u.x, y, u.z) <- Append(x, y, z)

and a goal of the form
<-Append(A_{m-1}.A_{m-2}.....A_1.NIL, B_{n-1}.B_{n-2}.....B_1.NIL, z_0)

where the A's and B's represent constants. A total of m Append subgoals will be called in the process of solving the top-level goal. The i'th unification ($1 \leq i \leq m-1$) will produce the unifier

$$\{u_i/A_{m-i},\ x_i/A_{m-i-1}.\ldots.A_1.NIL,\ y_i/B_{n-1}.B_{n-2}\ldots B_1.NIL,\ z_{i-1}/u_i.z_i\}.$$

If substitutions are applied explicitly then it can be seen that the i'th goal will have a total of m+n+1-i constants embedded in the terms of its first two arguments. It is readily verified that refuting the top-level goal will entail work of the order

$$\frac{m(m+2n)}{2}$$

in constructing the intermediate subgoals.

A structure-sharing implementation, on the other hand, will merely record the substitutions and have them available for later access. The process of recording the bindings will <u>not</u> involve the construction of terms: it will merely share the relevant parts of the supplied lists wherever necessary. The work in deriving the refutation under these circumstances is of the order m.

The inefficiency of the first scheme may be attributed to the fact that terms are constructed even though they may not be required. This is vividly illustrated in the above example by the 'fully detailed' second argument which is passed right the way through all intermediate calls without any demand being made on its actual contents. One might say that structure-sharing is based on application of substitutions by need whereas copying is based on the application of substitutions by availability.

We only consider a fully structure-shared implementation and this will be based on modifications of its conventional counterpart. This is not to deny that other approaches which mix structure-sharing and copying in certain ways are feasible - we merely do not consider them. There might well be a strong case for a hybrid scheme which constructs small terms and shares larger ones. (This would be analogous to many conventional computer architectures which, although based on instructions that reference data in store, make provision for so-called 'immediate instructions' that carry small amounts of read-only data explicitly.) Any proposal would have to be compared with others and in this context, we present our scheme as one based entirely on structure-sharing principles.

One way of implementing the required proof procedure is to divide the search tree into the set of branches descending from the root node and to investigate each branch independently. The data structures described earlier as being suitable for a backtracking interpreter could be carried over in a simplified form - for, of course, there would no longer be any need for information to do with backtracking. In particular, any variable's binding could be held in the activation record corresponding to the node at which the variable is introduced, rather than the one in which the variable is bound.

The difficulty of this approach is that because distinct branches of the search tree relate to distinct derivations, binding space for uninstantiated goal variables would have to be replicated whenever the search tree forks. In a structure-shared environment, it is not readily apparent which variables are bound and which are not.

We reject as too inefficient, the simplistic approach which, whenever the search tree forks at a node, provides a copy of the environment of bindings appropriate to that node for each of its children. Clearly, such a scheme would copy all existing bindings, with the consequent waste of time and store. Even the association list approach of Robinson and Sibert [40] implies the copying of a structure which is proportional to the depth of the search tree on each occasion that the tree forks and we reject this more optimised proposal on the same grounds.

Instead, we choose an approach which exploits the tree structure of the search space.

5.4 A NAIVE MODEL OF THE IMPLEMENTATION

We first give a naive approximation to the model we propose.

Each activation record corresponds to a node in the search tree. Each is a data structure of the form

ar(level, goal, clause, unifier)

where

'level' is the (unique) name of the activation record. As before, it is implicit in the storage address of the activation record.

'goal' names the goal which the activation sets out to solve.

'clause' indicates the clause being tried in order to solve the goal.

'unifier' is the unifier associated with the node.

The first three terms are carried over from the corresponding structure in the backtracking scheme.

The stratagem of holding a variable's binding in the variable's introductory - rather than binding - activation record is no longer available. Hence the replacement of the bindings and reset list of the conventional activation record by the unifier in ours.

Exploration of the branches can now be undertaken separately, for once a node is established, it cannot subsequently be altered. The backtracking algorithm given earlier may be adapted for exploration of the branches. The essential differences are:-

1. If unification at node N calls for the term binding of a variable introduced at node N' (N' must be N or a proper ancestor of N in the search tree), a search of bindings in unifiers associated with those nodes between N and N' inclusive is called for.

2. A unification failure terminates the branch computation.

5.5 A MORE PRACTICAL MODEL OF THE IMPLEMENTATION

The above scheme is naive because the simple ploy of searching a set of unifiers for a binding is, in general, orders of magnitude less efficient than the two-reference look-up described for the backtracking implementation. In a structure-shared implementation, looking up the binding for a variable is a frequently undertaken task and such a degree of inefficiency would result in an intolerably slow performance.

We now describe how the simple two-reference scheme in conventional use is adapted in the design of the Or-parallel proof procedure.

5.5.1 Representation of Bindings

Suppose a variable v is introduced into the computation at node N.

As previously explained, whenever the search tree forks, new instances of unbound variables come into existence, one instance per branch.

In this way, it may well happen that at a later time, some instances of v are bound and others are not.

We consider all bindings for the various instances to v to be registered, each binding indicating the instance of v to which it applies. In this way, the set of bindings for v form a register which we associate with the node at which v is introduced. Thus the activation record evolves to

ar(level, goal, clause, unifier, registers)

where 'registers' is an array of binding registers, one for each variable introduced by the clause referred to in the activation record. It will be seen that this arrangement is a generalisation of that for the backtracking interpreter where the corresponding array determines the unique (current) binding for each variable rather than a register of alternative bindings. The description, as it stands, implies some duplication because it indicates that each binding appears in one unifier and one register. In fact, this will not be the case in practice because the unifier is not required permanently - as will be shown in due course.

Let us now consider the evaluation of terms, an important aspect of any implementation. It was earlier pointed out that different branches of the search tree represent different derivations. Evaluation of a term needs to take account of which instances of its contained variables are involved. The instances are, of course, those corresponding to the derivation being pursued. To be specific, suppose, as in Figure 9 on page 71, that a binding for the variable 'v' needs to be looked up during unification at node N.

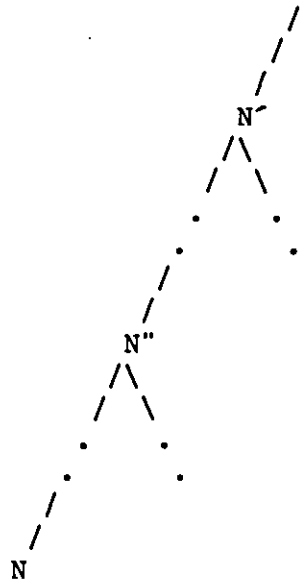


Figure 9.

Suppose also that v was introduced at node N' (an ancestor of node N). The derivation of interest is represented by the branch of the search tree passing through node N (and N') and extending back to the root. Hence to determine whether v is bound in the course of evaluation at node N, it is sufficient to determine whether v was bound by a unification at any node N" lying on the branch between N and the root node (in fact between N and N' inclusive) since these are the only nodes of any relevance to the derivation in question.

Perhaps the most obvious way of associating bindings with branches of the search tree is to associate each binding with the name of the node (the component 'level' in the corresponding activation record) which produced the binding and implicitly establish the tree structure by recording in each node (other than the root node) the name of its parent. (All ancestors of any given node may then be determined by finding the node's parent, its parent, etc. right back to the root node.)

We reject this approach on the grounds of inefficiency, for to test whether one node is a descendant of the other reduces to the problem of examining an arbitrarily long chain (the supposed descendant's

chain of ancestral activation records) and checking whether the other node has the same name as any one of them. The whole chain must be examined to determine that the answer is 'no' (assuming, of course, that this is the case).

Our design explicitly labels branches of the search tree and associates with each node the name of the branch on which it lies. Inclusion of the branch name in this way indicates to which derivations the node relates. It is the nature of the naming scheme itself, to be introduced shortly, that enables us to determine the ancestral relationship between seeking, binding and introductory nodes (N, N" and N' respectively) with which evaluation is concerned.

Thus the activation record structure takes on an extra term:-

ar(level, goal, clause, unifier, registers, branch)

5.5.2 Interrelation of Activation Records

If unification at a particular node N fails or is successful but all branches passing through N sooner or later lead to failures, then there is no point in keeping the activation record associated with N, since it cannot contribute in any way to a solution of the user's goal statement. Under these circumstances, we allow activation records to be deleted.

The deletion of all activation records corresponding to N's children is a sufficient condition for the deletion of N's activation record. In order to take advantage of this observation, we include two further items in the activation record data structure:-

ar(level, goal, clause, unifier, registers, branch, parent, children)

where 'children' is the set of activation record names (levels) of the node's children currently in existence and 'parent' is the level of the node's parent (or some indication of the root activation record).

We will presently describe how these arguments are used but first, a short digression is necessary in order to introduce some concepts in whose terms the description is given.

5.5.3 Processes and Messages

The specification of a scheme which supports parallel computation may be effected by identifying sub-computations that might proceed concurrently with one another and showing how those sub-computations are interrelated.

We will refer to such sub-computations as **processes.**

Examples of processes might be the sub-computations associated with nodes of the search tree. Such processes might reasonably be expected to manipulate data structures that represent nodes of the search tree (activation records) and we will find this to generally be the case: processes manipulate associated data structures. We will have more to say about this association later.

Processes communicate by means of **messages.** A message may be regarded as the triple:

<name of destination process, name of sending process, content>.

Messages imply processing: the processing that the destination process needs to do in order to take account of the content of the message.

In general, many messages may be sent to any given process but for reasons which will become apparent later, we impose the constraint that the processing required to put into effect the contents of two or

more messages sent to the same process is not allowed to overlap in time.

Thus at an abstract level, we see the computation being organised as a set of processes which modify 'their' data structures and communicate with one another through messages.

5.5.4 A Simple Computation

Before returning to the main text, we take the opportunity afforded by this interruption of giving a simple computation which will serve to exemplify several points raised in the immediately following sections.

The computation is a non-deterministic Append which finds all ways of splitting the list 1.2.NIL.

```
Append(NIL, x, x)
Append(z.u, v, z.w) <- Append(u, v, w)

<- Append(s, t, 1.2.NIL)
```

The search tree for the problem is given in Figure 10 below.

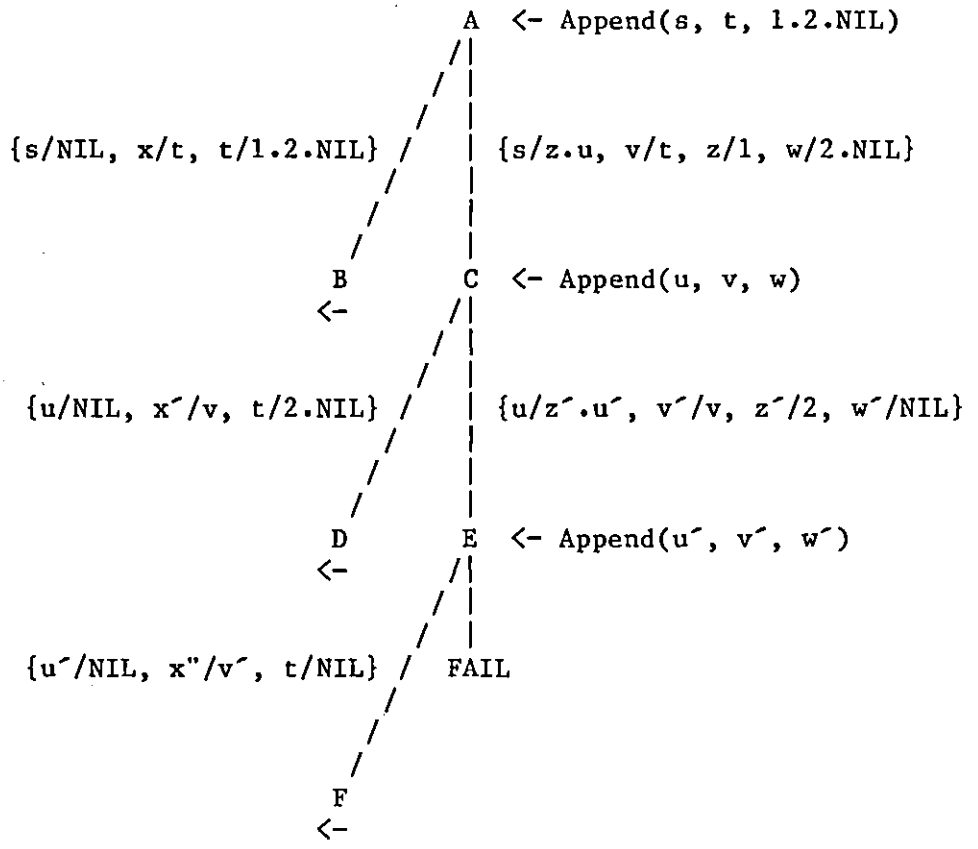


Figure 10.

5.5.5 Main Processes

We resume the central topic by describing here the sub-computation associated with the or-activation record of a particular node, N. Because such sub-computations are central to the scheme, we will term them **main** processes.

As in the backtracking implementation, an activation record will have been established and will include an indication of the goal and clause with which it is concerned.

The first task of the main process associated with N's activation record is to attempt the unification. If unification fails, N's parent main process is sent a termination message (the parent is known from the child's 'parent' argument) and N's main process arranges for the deletion of its own activation record and then terminates. On receipt of the termination message, N's parent eliminates N from its set of current children.

If unification is successful, a message to this effect is sent to N's parent main process. The reason for doing this is to enable N's parent main process to assign a branch name to N's activation record. We choose to assign branch names after unification is successful rather than before it is attempted because some restraint in the use of branch names is desirable. We will return to branch names and the timing aspect of their allocation presently.

Notice therefore that because the branch name along which N lies is not known until after unification is completed successfully, it is not possible to register bindings in the course of unification. Thus evaluation of a goal variable in the course of unification is a two part operation:- the unifier under construction has to be separately examined in addition to the variable's set of registered bindings. No constraint prevents these operations from being pursued in parallel - although our specific proposal does not consider this possibility.

Bindings in the unifier are registered once the branch name is known. Registration is essentially concurrent over the set of bindings in the unifier and is implemented by processes which have a register as their associated data structure. The description of registration is deferred.

After registration, the only information contained in the unifier that is not readily accessible from the relevant registers is an indication of which (goal) variables were bound in the course of that unification. This is the information carried in a conventional reset list. Thus after the bindings in the unifier have been registered, we may think of the unifier as a reset list which names goal variables bound in the unification. We will not elaborate on this any further here but will return to this use of a unifier later on.

Computation continues along conventional lines:- the next step is to ascertain whether the empty clause has been derived and if not, to apply the selection function in order to determine the next goal.

In the former case, the corresponding solution is extracted and we will show how presently.

In the latter case, the empty clause has not been derived and so the next goal is chosen (by the main process applying the selection function to the set of outstanding goals). The user's program indicates which clauses might be applied to the selected goal and the parent establishes an activation record for each such clause and notes the names of its children thus spawned. When it knows that registration of the bindings in the unifier just produced has been completed (exactly how will be described in the next section), the parent activates its child main processes by sending each a 'begin computation' message. We thus return to the starting point of this description.

Example

Consider the main process corresponding to node C in Figure 10 on page 75, which we will refer to as main[C]. Main[C] is established by its parent process, main[A] (as is main[B] which later fails, but we are not concerned with that here). On receipt of a 'begin computation' message, main[C] attempts the unification of Append(s, t, 1.2.NIL) and Append(z.u, v, z.w) which succeeds with unifier {s/z.u, v/t, z/1, w/2.NIL}.

Main[C] informs main[A] of the successful outcome and awaits the arrival of a branch name (it frees computing resources during this wait). Main[A] receives the notification of success and assigns to the activation record at C, ar[C], a branch name which we represent here as 'c'. Registration of the unifier is undertaken and the selection function applied to choose the next goal, trivially <-Append(u, v, w).

The program reveals that two clauses may be used to solve this goal and so two activation records, ar[D] and ar[E], are established by main[C]. When registration of C's unifier is complete,

the corresponding processes main[D] and main[E] are each sent 'begin computation' messages and they proceed with their processing independently of one another, in a manner similar to the above.

We can see that main[D] derives a refutation and when the solution is extracted, main[C] is sent a termination message from main[D] and removes the D-child from the children in ar[C]. Once main[E] sends a termination message, the E-child is also removed and the children argument in ar[C] indicates no children, so main[C] terminates by deleting ar[C] and sending main[A] a termination message.

Notice that our scheme allows unifications of the goal with the heads of all applicable clauses to be attempted concurrently via the independent child main processes. In this respect, it differs from the scheme proposed by Conery & Kibler [11]. It also allows for the concurrent registration of all bindings in a unifier and for these processes to run concurrently with the establishment of child activation records.

5.5.6 Registration

Register data structures are manipulated by corresponding registrar processes such that each registrar concerns itself with just one register.

The binding v(branch)/term, where v(branch) is the instance of v appropriate to the specified branch, is represented in v's register by the pair

<term, branch>

("term" represents, of course, the structure-shared pair <level, static term>). The function of v's registrar is to receive such pairs and incorporate them with others for the variable. Both the register and registrar are identified by the variable's name <level, static vari-

able> - i.e. if the variable's name is known, both may be accessed through it.

Let us suppose then that unification undertaken by some main process has succeeded, the parent main process has been informed of this event and has duly conferred on its child a branch name.

The sole purpose of including registration in the scheme is to make evaluations more efficient than they would otherwise be. Obviously, registration of a particular binding must be completed before an attempt is made to read that binding from the register. This implies that registration of those bindings produced in the course of unification at node N must generally be completed before resolutions at any of N's descendant nodes can begin.

Note that the registration of bindings in a unifier and the creation of child activation records are independent tasks. We exploit the parallelism this independence makes possible by arranging for the main process to devolve registration of its unifier to an associated unifier-reg process.

In fact, another related function needs to be considered at this time: that of bringing into existence registrar processes for variables introduced by the clause just used in unification. Stated precisely, the two tasks to be considered are

1. The initiation of new registrars for new variables and

2. The registration of bindings in the unifier - by devolution to the registrars concerned.

Some sequentiality is involved between registration of bindings and initialisation of registrar processes (specifically, the registration of input bindings) and we cope with this by arranging for the unifier-reg process to organise both the above tasks.

Thus the unifier-reg operates on the data structure

<unifier, registers, branch>.

where 'unifier' is the set of bindings of the form v/t made during the successful unification, 'branch' is the branch name conferred by the parent main process following the successful unification and 'registers' is the set of empty registers corresponding to the set of variables just introduced into the computation by the clause used.

In describing the implementation of the unifier-reg process, we make the obvious simplification of combining its two functions when dealing with input bindings. If the unifier contains the binding v/t, where v has just been introduced into the computation, then no other binding can ever be made for v. A registrar for v is thus superfluous and so rather than establishing one, the unifier-reg takes on the responsibility of inserting the binding v/t in v's register.

The computation required of a unifier-reg process is relatively trivial and is organised sequentially.

Each binding in the unifier is examined in turn. For input bindings, registration of the binding is undertaken by the unifier-reg itself. For an output binding v/term, a message, whose content is a request to register the binding <term, branch>, is sent to v's registrar process. Confirmation of registration will be sent by v's registrar to the unifier-reg process in due course. In the meantime, the unifier-reg proceeds with the next binding.

When the unifier has been fully scanned in this way, the unifier-reg process turns to the array of registers with which it has been provided (i.e. registers for the variables just introduced by the clause used). Some of these registers will still be empty - those for the variables not bound in the supplied unifier. For each such variable, a registrar process is brought into existence.

Once the unifier-reg process receives notifications that all registrations it delegated have indeed been made, it sends a message to that effect to its associated main process and then terminates.

The evaluation of terms containing variables whose bindings have just been registered will now be correct and main processes associated

with the child activation records may therefore be sent 'begin computation' messages.

Example

We look at the registration of bindings in the unifier at C. The unifier-reg[C] process, abbreviated here to unireg[C], is established by main[C] and operates on the structure

< {s/z.u, v/t, z/1, w/2.NIL}, [reg(z), reg(u), reg(v), reg(w)], c>

where the second argument is an array of empty registers.

Unireg[C] calls on the s-registrar to register the binding <z.u, c>. Unireg[C] itself sequentially registers the bindings for the variables v, z and w. It determines that u is a new variable that has yet to be bound and so establishes the u-registrar whose data structure is the empty u-register. On receipt of a message indicating that the binding for s has been incorporated, unireg[C] informs main[C] that all bindings have been registered and terminates.

We now turn to a description of the registrars themselves.

Let us suppose that a message requesting the registration of a particular binding has been received by the registrar for the variable concerned.

The binding is registered in accordance with the data structure chosen to represent registers. This might, for example, be a set or we may define an ordering on branch names and exploit it by using an ordered data structure such as a list or tree to represent the register. These considerations are left until later.

Once the binding is incorporated in the register, a message to this effect is sent to the unifier-reg process which issued the request.

This completes the description of registration.

Example

Suppose that the unireg[B] and unireg[C] processes are operating concurrently with one another. The s-registrar will, in due course, receive messages from unireg[B] and unireg[C] respectively requesting registration of the bindings <NIL, b> and <z.u, c>. The t-registrar will be sent a message (by unireg[B]) requesting incorporation of the binding <t/1.2.NIL, b>. Unireg[B] and unireg[C] will be expecting 2 and 1 confirmatory messages respectively and on receipt of the appropriate number, they send their associated main process a termination message and terminate.

5.5.7 Solution Extraction

We assume that the user specifies, as part of the problem he submits, which goal variables are of interest to him. A solution is then the set of bindings for those variables (and variables nested in the term components of those bindings etc.) which apply at a node that derives the empty goal statement.

Solutions are conveyed to the user through a unique answer process, accessible from all main processes. Thus on deriving the empty clause, a main process sends the message 'I have an answer' to the answer process and then awaits acknowledgement that the answer has been extracted. The answer process has available to it the source of this message and so is able to determine the branch along which the refutation was made. Consequently, it is able to interrogate the registers for bindings appropriate to the refutation and thereby extract the solution.

When it has finished this extraction, the answer process sends an 'answer extracted' message back to the main process which called it. This main process then sends a termination message to its parent main process, deletes its activation record and terminates.

This completes the outline description of the proposed implementation.

5.5.8 Branch Names

We have already discounted, on grounds of efficiency, the scheme that implements branch names implicitly by means of ancestral node chains. Instead, we choose to represent branch names explicitly by means of bit patterns and will show that the determination of whether one branch is an extension of another reduces to the problem of determining whether one bit pattern is an extension of another. (The branch name on which the root node lies will, by convention, be the empty bit string - i.e. the bit string of length zero.) Clark and McCabe use a similar scheme to control co-routining in IC-PROLOG [7].

Having made this decision, it is important that branch names be kept short and this is the underlying reason for waiting until unification is complete before providing a branch name. To do otherwise, particularly when dealing with a large relation, could prove wasteful. We propose two branch naming schemes, the n-ary and binary schemes.

5.5.8.1 N-ary Branch Naming Scheme

Let us suppose that unification at node N succeeded and that m activation records were spawned as a result. In the n-ary scheme, branch naming cannot, in general, commence until unifications undertaken in all spawned main processes are complete and the number of successes is known.

Let us suppose that the branch name at node N is the string of bits $b_1 b_2 \ldots . b_j$ (of length j) and that n of the m unifications are successful.

If n is 1, the child's branch name will be the same as the parent's. (Note that the case of m=1 - i.e. only one unification was attempted - is an exceptional case that allows the child's branch name to be allocated before unification. We will have more to say about this and related cases later.)

If n is greater than 1 then the n children whose unifications were successful may be counted, each associated integer being expressed in k-bit binary form, where k is the smallest integer not less than $\log_2(n)$. (The order in which children are counted is immaterial.) The branch names corresponding to the n children will then be the j+k length bit patterns formed by appending the child's associated k-bit number onto $b_1 b_2 \ldots b_j$.

For example, if the branch name at node N is 1011 and 3 child unifications succeeded then these may be counted by the 2-bit integers 00, 01 and 10 and the resulting branch names will be 101100, 101101 and 101110 respectively.

The scheme allocates branch names unambiguously, a conclusion which is easily verified by an induction on the incremental lengthening of branch names, which we give below for the sake of completeness.

Clearly, the extensions which count the successfully unified children of some given node are all distinct from one another. Let us assume the inductive hypothesis - that all branch names are distinct - holds at some stage of the computation and let us consider the immediately following step of allocating branch names to the successful children of some node N.

Their branch names will be different from one another by virtue of the differing extensions they have over N's branch name.

Their branch names will each differ from other (previously existing) branch names because by the inductive hypothesis, the initial part of their names (N's name) differs from all previously existing branch names.

Finally, the base case of the hypothesis holds because initially, when the root node is grown, there is only one branch of the search tree.

5.5.8.2 Binary Branch Naming Scheme

The binary branch naming scheme is a slight modification of the n-ary scheme insofar as there is no longer a requirement that the outcomes of all child unifications be notified to the parent before the first branch name is assigned: the cost is some wastage in the length of branch names.

We suppose again that the unification at node N succeeded and that m main processes were spawned. As before, if m=1 there is no need to wait since the child node will lie on the same branch as N.

Suppose then that more than one main process was spawned. Notification of unification outcomes is awaited at N as before but now, as soon as two successes are notified, the activation record for one of them is given N's branch name appended with a zero bit. (The existence of two successes is the minimum necessary to determine that the search tree forks and hence new branch names are needed.)

If there are no other successes, the second activation record is given the branch name of N appended with 1. The net result in this instance is the same as that for the n-ary naming scheme; the only difference is that, generally, a shorter overall delay is involved.

Alternatively, if a third success is notified, the waiting main process is given N's branch name appended with the bits 10. Just one main process is now waiting (as before) and further progress is made in a manner analogous to that described immediately above, the outcome depending on whether a fourth success is notified or not (etc.).

Thus in the binary scheme, branch names are formed by lengthening the name of the parent's branch by adding bits 0, 10, 110, ..., 111...10, 111...11 .

Once more, the branch name extensions of children descended from any node are distinct from one another and this is the crucial property required to demonstrate that all branch names are unique. We do not repeat the proof for this modified name allocation scheme.

5.5.8.3 Integration of N-ary and Binary Branch Naming Schemes

It will be seen that the n-ary and binary schemes described above offer the familiar trade-off between space and time. For n branches ($n > 1$), the n-ary scheme needs $O(\log_2(n))$ bits for the extension to the branch name, the binary scheme needs $O(n)$ bits. The penalty for the more compact scheme is the need to wait for all unifications to be complete; the binary scheme releases branch names at the same rate as successful unifications are notified - albeit lagging one behind.

Under circumstances of light loading, where resources are underutilised, the shorter overall delay afforded by the binary scheme may be desirable and the waste of resources it entails perfectly acceptable. If, on the other hand, the machine is adequately loaded, there is no point in reducing the delay in any isolated part of the overall computation - for by assumption, other parts of the computation may continue. We thus see the n-ary scheme as being of more significance than the binary.

It should be clear that the two schemes are quite compatible: it is possible to alternate between them at any given node.

Thus one could envisage the following situation where 6 successes are notified to node N, the first two during a period of little activity, the remainder at a busier time. (The bit patterns shown are extensions of N's branch name).

0	(first binary)
10	(second binary)
1100	(These last four branch
1101	names are not assigned
1110	until all successes
1111	have been notified).

Exclusive use of the binary scheme would result in longer branch names (parent's branch name extended by 0, 10, 110, 1110, 11110, 11111); exclusive use of the n-ary scheme would result in more compact branch names (extended by 000, 001, 010, 011, 100, 101) at the cost of delaying progress on the first two children.

5.5.8.4 Pre-allocation of Branch Names

We pointed out that if only one clause can be applied to a chosen goal, the branch name may be conferred on the corresponding activation record before unification is attempted and in this case, the name will be the same as that included in its parent's activation record.

This modification may be extended to the case of a conditional where it is known in advance that at most one of the clause heads will unify with the chosen goal. All alternative activation records will be given the parent's branch name but in this case, more than one successful child unification is treated as an execution error.

More complex is the case of "don't know non-determinism" [28] where a goal is known to have no more than one solution and where determinacy is established not through the head but through predicates in the body. To see why, consider the following goal and procedure set

```
<- P(A, z, z)

P(x, y, B) <- Q(x, y)
P(u, v, C) <- R(u, v)
```

and suppose that it is known that for any given x, only one of Q(x, y) or R(x, y) holds. The heads of both clauses will unify with the given goal but if both child nodes are given the same branch names then the alternative bindings for z, z/B and z/C, will be registered with the same branch name and clearly this will lead to confusion when the execution of the Q or R subgoals needs to access a binding for z (via the bindings y/z and v/z).

We describe below a simple modification of our scheme which will overcome this deficiency.

The difficulty is removed if registration of bindings is delayed until determinism is established, that is, until there remains just one child of the main process whose selected subgoal is known to have a single solution, all other children having been deleted through failure. This then implies the need, when searching for a variable's binding during a later unification, to not only seek that binding in the unifier under construction (as well as the variable's register) but to also look for it in past unifiers still awaiting registration. This does, of course, detract a certain amount from the advantages of registration - which was introduced to minimise such searching.

We will not pursue this modification any further here.

Another possibility to consider is that of "don't care non-determinism" [28]. Here, we are presented with a goal and we know that for this particular goal, any of the clauses in the corresponding procedure set may be used to solve it: the result will always be the same. In this special case, only one of those clauses need be selected and the branch name of the resulting child node will be the same as that of its parent. As with most forms of 'intelligent' computation, considerable overheads may be involved in determining that the special case applies and this must be offset against any savings made through ignoring the non-selected clauses.

5.6 REGISTER STRUCTURE AND MANIPULATION

5.6.1 Structure

We stated earlier that an entry in the register of bindings for any given variable is a pair of the form <term, branch> and we proceeded to fix the branch name as a bit pattern. Both n-ary and binary branch naming schemes guarantee that node y is a descendant of node x if and only if x's branch name, of length n bits, is identical to the first n bits of y's branch name. In this event we say that y's branch name descends from x's:-

Node-descends(x, y) <-> Branch-descends(branch(x), branch(y)).

This equivalence is exploited in two distinct ways.

1. When seeking a binding in a variable's register, both branch names are known and the equivalence determines whether or not the 'seeking node' descends from any of the 'binding nodes'.

2. Every node at which a given variable is bound descends from the node at which the variable is introduced. The equivalence highlights the redundancy in storing in each binding the common part of the binding node's branch name; all that is needed is to store the extension of the longer name over the shorter. Thus if v is introduced along the branch named 1010101, any binding for v must be made along a descendant branch. For example, the two bindings for v, <A, 101010100> and <B, 10101011>, may be stored as <A, 00> and <B, 1> respectively. The resulting savings in resources should be evident. Note that if an input binding is being registered (by the unifier-reg process), the extension will be represented by the empty bit string.

Example

We give below the full set of registers arising from computation of our earlier Append example. Here, the names of the branches associated with nodes A, B, C, D, E and F are respectively ´.´ (the empty bit string), 0, 1, 10, 11 and 110. Only the branch extensions are recorded in the bindings.

s (introduced at A (.))	{<NIL, 0>, <z.u, 1>}
t (" " ")	{<1.2.NIL, 0>, <2.NIL, 10>, <NIL, 110>}
x (introduced at B (0))	{<t, . >}
z (introduced at C (1))	{<1, . >}
u (" " ")	{<NIL, 0>, <z´.u´, 1>}
v (" " ")	{<t, . >}
w (" " ")	{<2.NIL, . >}
x´(introduced at D (10))	{<v, . >}
z´(introduced at E (11))	{<2, . >}
u´(" " ")	{<NIL, 0>}
v´(" " ")	{<v, . >}
w´(" " ")	{<NIL, . >}
x"(introduced at F (110))	{<v´, . >}

We now consider exploitation of an ordering on branch names. Clearly, if the bindings in a register are ordered, a search for a binding which applies at a specified branch need not, in general, be exhaustive. The ordering is defined as follows:-

Let B_1 and B_2 name two branches.

$B_1 < B_2$ if the first bit position at which B_1 and B_2 differ has B_1´s bit as zero.

This ordering may be extended in the obvious way to two extensions of the same branch name.

It is clear that for any two distinct branches, B_1 and B_2, one, and only one, of the four conditions

B_1 descends from B_2

B_2 descends from B_1

$B_1 < B_2$

$B_2 < B_1$

holds.

When concerned with the ordering of the bindings in a register, we need not take into account the possibility that the branch name of one binding descends from that of another because the binding applying at the earlier branch would also apply at the later one and this would preclude an alternative binding. Thus the above ordering may be considered a total ordering on the restricted domain of branch names in a register.

As regards the actual structure of the register, we put forward two proposals, the ordered chain and the ordered binary tree. We reject storing bindings in a mass associative memory on the grounds of impracticability since there appears to be little prospect of such memories being cost-effective in the short to medium term, the timescale of primary interest in the Or-parallel proof procedure.

Chain and tree structures share the property that searching a small register and inserting a new binding are both efficient operations. A tree structure is more expensive on storage but only needs logarithmic time to be searched, potentially a significant advantage for large registers. The most obvious disadvantage of tree structure is that deletion of a node in the tree is not, in general, a trivial operation* and might involve considerable processing. We will consider this disadvantage after the next section, which discusses the manner in which registers might be manipulated.

5.6.2 Insertions

We need to impose the restriction of having just one registrar per variable because otherwise the administration of the register would be

* in the context of a parallel implementation

far more complex. To appreciate this, note that data structures can only be 'held together' (in non-associative memory) by juxtaposition or reference pointers and the former possibility is ruled out for a register because the amount of storage eventually needed is not known at the time the variable is introduced into the computation. Modifying such structures, unfortunately a necessary evil in practice, must therefore be done by modifying the reference pointers, and this needs to be carried out in a controlled manner.

If, for instance, registers were implemented as chained lists, some sort of interlock would be needed to prevent any attempt to 'simultaneously' chain two new bindings between the same pair of existing ones. Provision of such a mechanism would not be without its overheads and in all probability would result in slower overall performance than that expected from our single registrar proposal - although to be sure, simulation or experimentation would be needed to confirm this. At this point in the investigation, we do not regard the inability to register more than one binding for a given variable at a time (leaving others waiting until incorporation is complete) as a serious problem and so we accede to the constraint of having a single registrar per variable.

However, we would prefer not to have to impose a similar constraint when it comes to reading registers, for reading a binding is generally done quite frequently. Since reading a register does not alter its structure, there is some hope that registers might be arranged in a manner which allows any number of processes to access them at any instant of time. Certainly, processes which read a register will not interfere with one another, the only possible source of interference is between such a process and the registrar process and we now consider how such interference might arise.

Of course, ensuring that registration of all the bindings in a unifier is completed before releasing 'begin computation' messages to child main processes is a sufficient condition to guarantee that there is never an attempt to access a binding awaiting registration. Therefore interference can only occur if the registrar is in the process of adding an entry and the resultant modification to the structure upsets a process seeking an unrelated entry.

In fact, such interference does not come about and we demonstrate why by considering the ordered chain representation of a register. A similar argument holds for the ordered tree representation.

Refer to Figure 11. Suppose the registrar is in the process of inserting a binding B between the bindings A and C (N.B. a modified argument holds if A and/or C is null, that is, B is being inserted at the beginning and/or end of the chain). If a, b and c are the branch names contained in the bindings, we have a < b and b < c.

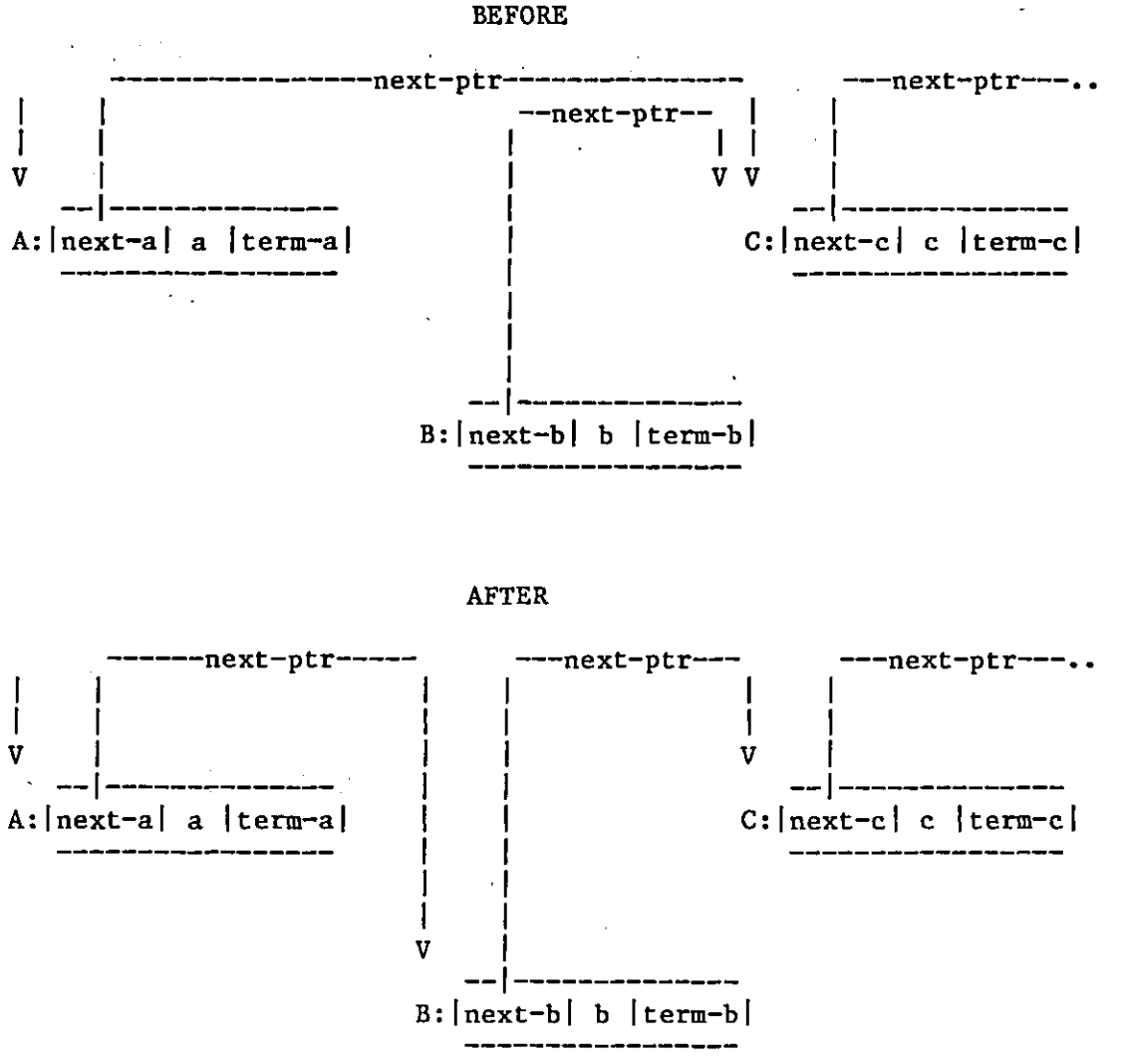


Figure 11.

The entry B is prepared for insertion and the content of A's 'next-pointer', next-a, is copied to next-b. At this point in time,

the state of the chain is as illustrated in the BEFORE part of the figure.

Suppose now that an independent process is seeking a binding which applies at a node whose branch name is x and it has already been determined that a < x.

The transition of the register from the BEFORE to the AFTER state is accomplished by a single machine instruction, "write", which assigns B's storage address to A's reference pointer, next-a. We need to consider what effect this transition will have on the independent searching process and for this purpose, we suppose that the search is just about to access the next entry, referenced in next-a, by the instruction "read".

"write" and "read" cannot access next-a simultaneously; the hardware physically prevents this. Thus whether B or C is examined next by the searching process depends on whether next-a is accessed by "write" or "read" first.

If B is chained in before next-a is read ("write" is first), the next comparison done will be between the bit patterns representing x and b. Because x is not descended from b, we have either x < b or b < x.

Case 1. x < b. If x < b, the sought binding is absent. But if B had not been chained in, the comparison x < c would have been done instead and because of the transitivity of '<' and the well-ordering of the register, x < c would have been established and so the same conclusion, viz. the sought binding is absent, would have been reached.

Case 2. b < x. In this case, the next entry in the chain, C, will be examined and the outcome will be as though B had not been chained in, i.e. C had been examined immediately after A.

It follows that insertion of a new entry cannot interfere with any search and so may be done independently.

5.6.3 Deletions

The possibility of deleting a binding from its containing register arises once it is determined that the node at which the binding was made does not relate to any refutation. This corresponds to backtracking in the conventional implementation when bindings are destroyed either implicitly by destruction of their containing activation record or explicitly by means of the reset list.

We have already described how deletion of an activation record may come about: essentially when all its child activation records have been deleted. Such deletion would automatically destroy the registers held in the activation record (terminating any registrar processes associated with them) and is analogous to implicit destruction of bindings in the conventional implementation.

Analogous to explicit destruction in the conventional implementation - which destroys bindings for goal variables - is the selective deletion of bindings from registers. Such deletions cause difficulties in our scheme and to see why, we need only consider the situation in Figure 11 on page 93 when we wish to remove the binding B from the chain in the AFTER state.

It is easy enough to change the pointer in next-a to point at C. The problem arises if this is done while a searching process is looking at the entry B (which of course is no longer required) and the timing is such that the unchaining of B is done, B is garbage-collected and its storage is re-assigned and then overwritten, all before the searching process gets around to comparing its supplied branch name with what it thinks is in the store previously occupied by B. Although the above circumstances are highly unlikely, they rule out deletion as described.

We are left with the choice of ignoring the savings in storage utilisation which such deletions make possible or designing register access in a way that prevents deletion of an binding from taking place if that binding is being accessed by a searching process.

We reject the latter option on the same grounds as we rejected the ability to simultaneously insert more than one entry at a time into the register - the necessary overheads in providing interlocks would in all probability more than offset any savings made.

The possibility we leave ourselves with, namely that of not deleting unwanted output bindings, does not, of course, affect the correctness of the proof procedure: it merely wastes storage and processing time in performing redundant comparisons. On the other hand, it does remove the need for a reset list and so we may tidy the earlier description by stating that once bindings in a unifier are registered, that unifier may be garbage-collected since its residual function as a reset list is no longer needed.

Our feeling in the absence of suitable simulation is that such waste would not typically precipitate a catastrophic failure because each binding takes so little storage. In the ordered chain representation, a binding is made up of three references (next-in-chain pointer and two pointers representing the term) plus the branch extension. Each reference might be 3 bytes (24 bits), the branch extension is, of course, variable but unless the search tree is very bushy or is less bushy but variables are frequently bound a long way after their introductory node, it too might typically occupy no more than 3 bytes, giving an overall typical size of 12 bytes.

Finally, we point out that our election not to delete entries from registers removes the basic objection to tree structure and this seems the most promising overall representation. (The tree representation needs an extra reference pointer per entry, resulting in a typical binding size of 15 bytes.) Thus there are compensations in choosing not to delete redundant bindings from registers and the net effect of opting for this approach would, as ever, need to be investigated by means of simulation.

5.7 DATABASE APPLICATIONS

In a serious database application, one may suppose that vast

relations of ground assertions exist in the program. It may not be suitable to treat such clauses in the same way as those for relations more compactly expressed. For instance, one may like to take advantage of the availability of special purpose associative searching hardware - for example CAFS [31] - which is fast at searching such ground relations.

Here, we will be content in stating how such an attached 'searching engine' may be used to supply all solutions of a given atomic goal and we show the modifications to our proof procedure needed to achieve this end.

We assume the user's program contains clauses of the form

$$P(t_1, \ldots, t_n) \leftarrow \text{"Consult Search Engine}(P(t_1, \ldots, t_n))\text{"}.$$

Such clauses may be additional to ordinary Horn clauses for the named relation P.

Only main processes are modified. Assuming a goal which potentially matches the head of such a clause is selected by some main process (which we will term the grandparent main process), this distinguished clause is seen as merely another way to solve the goal and so an activation record and main process corresponding to it are established in the usual way.

The main process (which we will term the parent main process) attempts unification as normal and a failure is treated in the usual way. In the event of a successful unification, however, the goal atom is fully evaluated, that is all variables are explicitly substituted by their bound terms. Note that the normal objection to applying substitutions, namely the copying of complex data structures, does not usually hold in database applications. The instantiated query is then sent to the searching engine.

We assume that the searching engine produces a set of solutions and these are returned to the querying main process (i.e. the parent).

The parent then establishes a child activation record for each solution it receives in such a way that solutions appear as unifiers - i.e. as though it was the child that had performed the unification with the corresponding ground clause. The child activation record also has allocated to it a branch name at this time.

Processing of each such child is arranged to start at the point where registration of the bindings is about to be made. 'Begin computation' messages are then sent to the child processes.

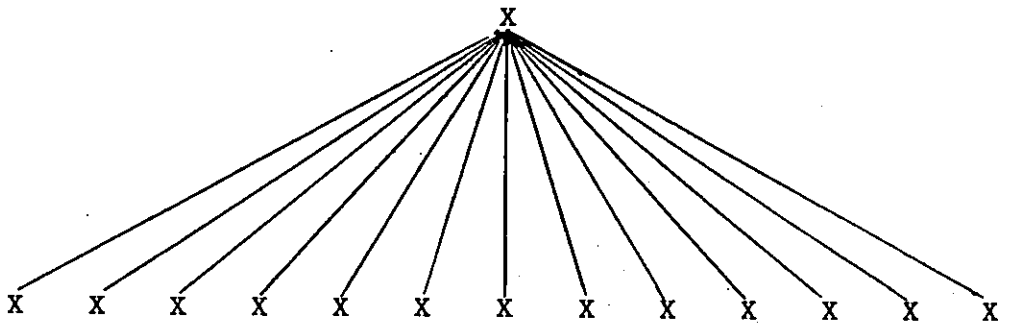
The modification may be summarised by considering what would happen if the database clauses had been explicitly expressed.

In this event, the top-level main process (the grandparent) would have established many children and each would have attempted its unification and informed their parent of the outcome. Successful children would, after receiving their branch names, have continued by establishing unifier-reg processes to register the bindings in their unifiers.

In the modification, the process corresponding to the "Consult Search Engine" clause takes on the task of performing all unifications and it only establishes child processes for the successes - i.e. the query solutions. Branch names are pre-allocated to these child activation records and the corresponding main processes begin at the point where they establish unifier-reg processes to register the bindings in the unifier.

The modification is illustrated in Figure 12 on page 99.

EXTRACT OF SEARCH TREE (NORMALLY EXPRESSED PROGRAM)



EXTRACT OF SEARCH TREE (PROGRAM INCLUDES "SEARCH" CLAUSE)

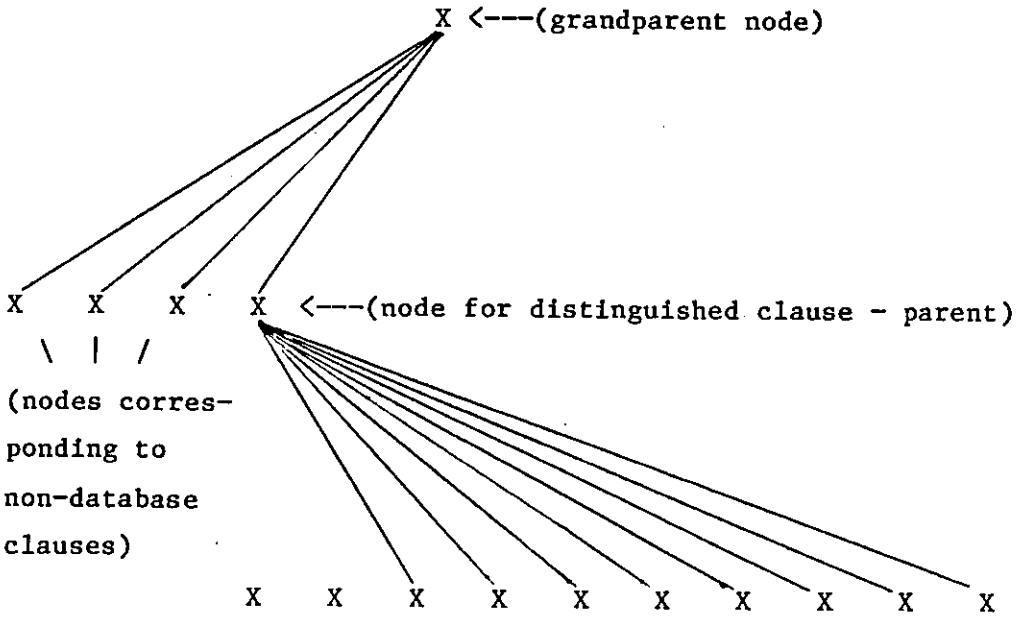


Figure 12.

5.8 ARCHITECTURE

We now turn to the consideration of a suitable architecture for our proof procedure. We point out once again that the Or-parallel scheme is seen as being essentially a short - medium term proposal and the lowest level of our design, the architecture, is put forward with this in mind.

5.8.1 Requirements

We take the opportunity of stating here the principal components already fixed in the design.

1. A conventional memory in which to hold data structures.

2. A message transfer system to allow inter-process communication.

3. A means of distributing work.

Additionally we need a fully distributed system - i.e. we want no central bottlenecks - and this requirement guides the design.

5.8.2 Memory

Our design has no place for a centralised memory such as that found in [12] because the traditional von-Neumann processor-store bottleneck will result on addition of sufficiently many processors. The alternative is a distributed memory, one segment of memory per processor.

A distributed memory may be implemented in one of two distinct ways - locally or globally.

In a **local** implementation of a distributed memory, we associate each segment with its processor in such a way that the only means of accessing that memory is through the processor itself. A number of architectures take this form, notable amongst them being ZMOB [36].

In a **global** implementation of a distributed memory, we allow any processor access to any segment of the shared memory without involving the processor associated with the segment and rely instead on hardware arbiters to cope with multiple simultaneous accesses to the same segment.

A local organisation is attractive from the implementation point of view for it then becomes very easy to control access to any particular part of store: all such accesses have to pass through the same processor and this constraint can be used to prevent attempts at simultaneous updates etc.. However, the greater the storage access burden, the more embarrassing a local organisation becomes, for any processor is then liable to frequent disturbance in order to satisfy others. Unfortunately, our structure-sharing scheme, so heavily dependent on accessing the same storage from an arbitrary number of processes would seem to rule out this possibility and in the absence of simulation, we will assume that this is indeed the case. Consequently, our architecture will be based on a global implementation of the shared memory. It will consist of processing elements (PE's) and Figure 13 on page 102 illustrates the first approximation to the structure of a PE. The precise nature of the connection between PE's is left unspecified for the moment.

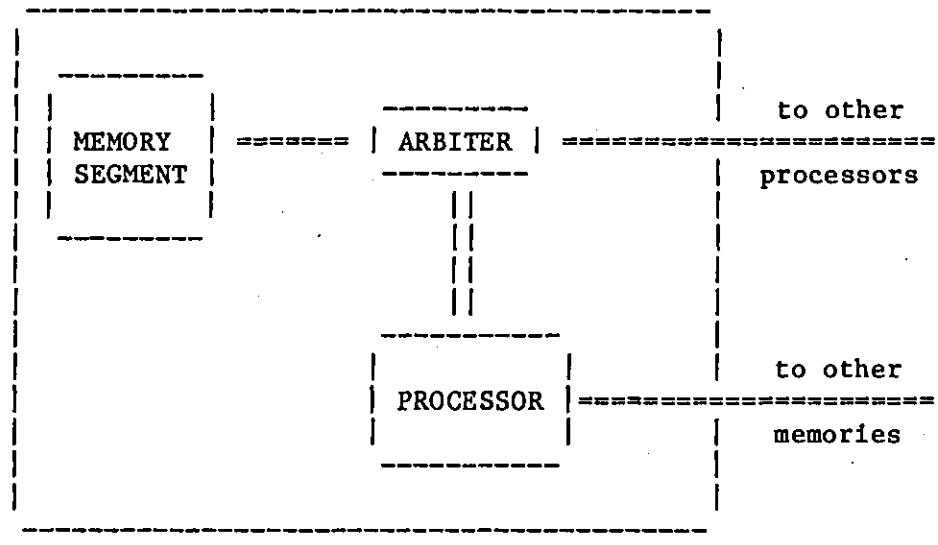


Figure 13.

Notice that a processor may access the segment of memory included with it in the PE without placing any load on the external processor-memory network. It is reasonable to expect that such accesses will be faster than accesses of external segments - although by how much rather depends on the nature of the network.

5.8.3 Packets

We introduced the concept of a process by stating that it identified a part of the overall computation which might proceed in parallel with other processes. Implicit in this notion is that some sequentiality in the execution of processes is involved - for otherwise increased parallelism is trivially obtained by further subdivision of the computation.

Processes communicate through messages and these messages imply work, the work which the receiving process needs to perform in order to put the message into effect. The sequentiality constraint we

impose is that only one of the outstanding messages received by a process may be put into effect at any one instant of time.

Thus we do not care about the order in which a registrar process adds bindings to the corresponding register. We do, however, insist that bindings are only added one at a time - otherwise a corrupt register would result, as was pointed out at some length earlier.

Similarly, we do not care about the order in which the unification outcomes of sibling main processes are notified to their parent. We do, however, insist that such notifications are handled one at a time. Were this requirement to be violated - for instance, if two distinct children both fail their unifications and the list of current children, held as an argument of the parent's activation record, is not updated sequentially - the proof procedure would be incorrect.

We may formalise the constraint of sequentiality by introducing the notion of a **packet**. We envisage a process as being implemented through a stream of packets of work, non-overlapping in time, each packet representing the computation implied by the contents of the message that gave rise to it. Later we will show exactly how packets are implemented.

An example of a packet might be the computation necessary to incorporate a new binding in a register. A stream of such packets implements a registrar process.

5.8.4 Distribution of Work

Viewing a process as being implemented by a stream of packets naturally raises the question of whether processes are to be run to termination by the processing element which first took them on or whether the processing elements are to be regarded as equal computational resources - packet processing agents in the nomenclature of Darlington and Reeve [13] - each capable of operating on any process.

By restricting each process to a chosen PE, it is reasonable to expect a significant reduction in the volume of processor to shared-memory communication - for the data structure associated with each process would be stored in the PE's own segment of memory. This would speed up computation, as indicated earlier. Moreover savings in the amount of storage used to hold program clauses could be made and we explain how in due course.

On the other hand, the alternative scheme is more flexible, for it avoids problems of local overloading where a PE has too much work. Furthermore, whenever the distribution of packets is uneven with respect to PE's, a loss in the realisation of available parallelism will result since some computing resources will lie idle whilst others will be overloaded.

Our view is that the spirit of parallel computation is best served through the adoption of the most flexible scheme - that of regarding processing agents as equal computational resources - but that practical considerations may temper this view to accommodate specialist PE's or groups of PE's and we will say how later. In this respect, our proposed implementation differs fundamentally from the scheme put forward by Conery & Kibler [11], who establish process-PE links by insisting that each process is run to completion on the PE that first accepted the process.

Our proposal to view PE's primarily as agents of computation has ramifications in the way that inter-process communication might take place and we now consider this aspect of the design.

5.8.5 Message Communication

In what follows, we will term a process active if a packet for that process is currently undergoing computation and suspended otherwise. Thus a suspended process is one which is awaiting the receipt of a message whilst an active one will have received one, possibly with further messages outstanding and awaiting completion of the packet.

By choosing to divorce processes from PE's, we seem to be disregarding the simple communication device that implements inter-process communication as inter-PE communication. In fact, this is not the case, as we will show later. First though, we consider a centralised process communications scheme, as illustrated in Figure 14.

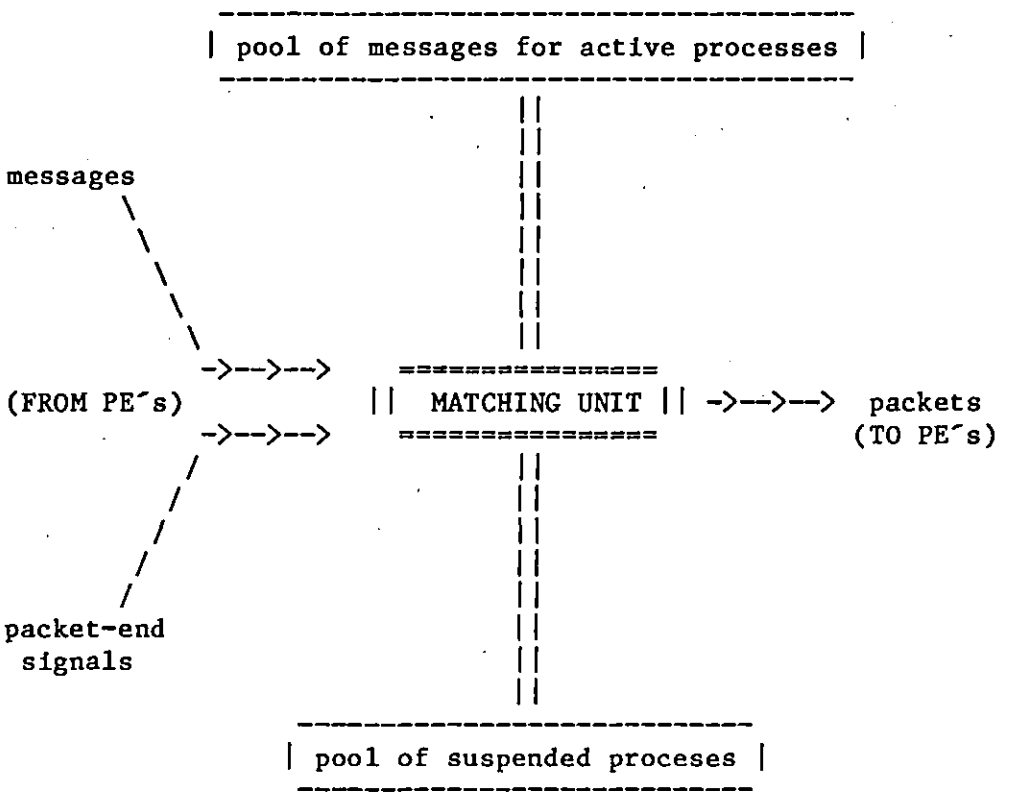


Figure 14.

Here, the matching unit maintains a pool of suspended processes and a pool of messages for active processes. It accepts messages from PE's and packet-end signals, also from PE's, the latter indicating that the packet for the named process has completed computation and hence that the next message for the process may now be formed into a packet and released for computation.

The matching unit operates as follows:-

An incoming message causes the unit to determine whether the process named as the destination of the message is in the pool of suspended processes. If it is, the resulting packet is dispatched. Otherwise, the message is pooled.

An incoming packet-end signal causes the unit to seek from the message pool a message addressed to the process named in the packet-end signal. If one is found, a new packet is formed and dispatched. Otherwise, the process is added to the pool of suspended processes.

Although such an approach seems attractive, it does introduce a central element into the architecture. Our scheme decentralises the matching unit in the following way:-

The modification exploits the association between a process and its data structure. Essentially, it takes account of the fact that any data structure is accessible through a single, well-defined memory location, its (implicit) name. Since we associate a segment of store with each PE, it follows that any data structure, and hence process, may be associated with that PE if its location lies within the PE's segment of memory. In this case, we say the process is based in the PE. We emphasise once more that we do not, in general, insist on a process's computation being performed by the PE in which it is based.

We see a matching unit, as depicted in Figure 14 on page 105, being incorporated in each PE. The matching unit is concerned with processes based in its PE and with messages to those processes. It also receives packet-end signals concerned with processes based in its PE. Note that packet-end signals take the same route as messages insofar as they are emitted from one PE's processor and received by another PE's matching unit. The distinction between them is that packet-end signals are intended for the matching unit itself whilst messages are passed via the matching unit on to the destination process. In the interest of brevity, however, we will refer to both as "messages" when discussing communication, it being understood that in this context, packet-end signals are included under the term. Whenever we wish to talk about messages in the previous sense, we will refer to them as "process-process messages".

We are almost ready to present a global picture of the architecture but first, we need to be more specific about how packets are sent out by one PE and received by another.

The obvious medium for this communication is the same as that used for communicating messages, provided that the medium allows this type of message to be broadcast in a way that prevents more than one PE accepting any given packet. We will call such messages **packet-start signals** and as one might expect, we also include them in the umbrella term "message" when using that term in the context of communication.

Following Farrell et al. [16], Darlington and Reeve [13], Rieger et al. [36] and others, we find a ring implementation of the communications medium one well worth investigating.

Essentially, the ring may be regarded as a continuous conveyor belt, with messages placed in circulating slots. Each PE has a single connection to the ring, a window, through which it has access to the ring. If the slot opposite the window is empty at any given time, the PE is allowed to place a message in it. If it is full, the PE is allowed to read the message, and, on the assumption that the message has arrived at its destination, to remove it, thereby leaving an empty slot once more.

Provided the ring allows messages to be broadcast - as the ZMOB ring [36] does - it will be seen that the ring communication medium satisfies our requirements.

Figure 15 on page 108 gives an outline of the proposed global architecture.

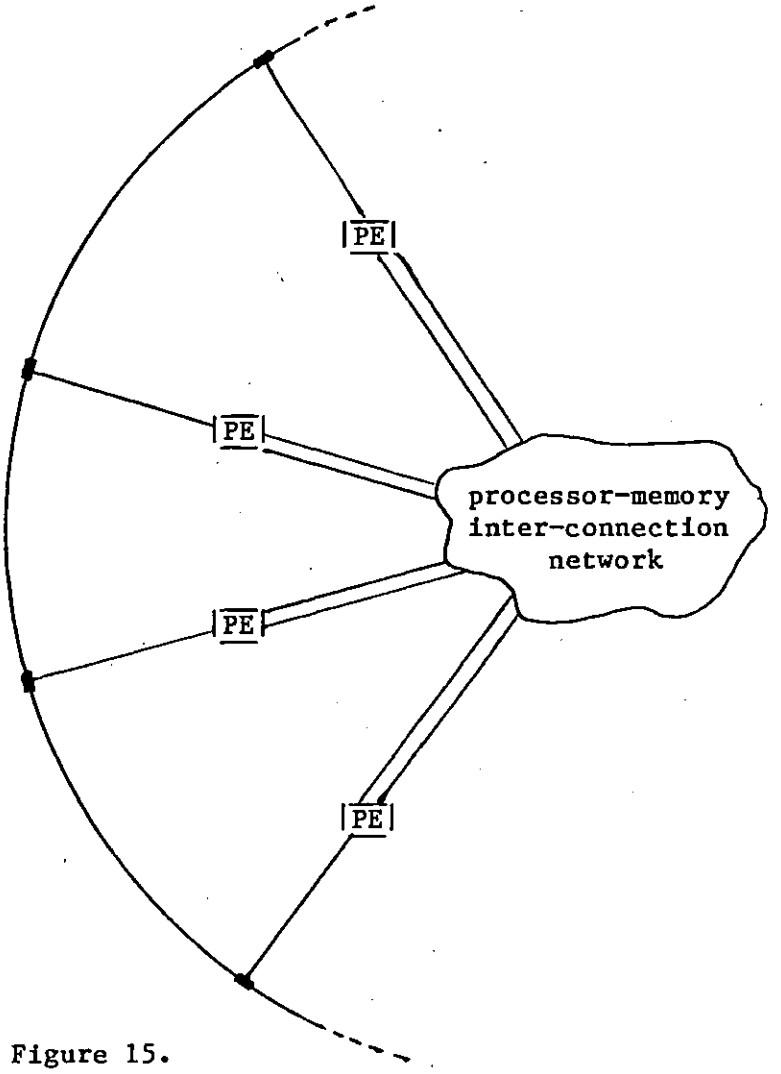


Figure 15.

Note that the connections between the processors and segments of global memory external to their PE have yet to be specified and this we will do shortly. We emphasise here once more that we require direct processor-memory connections to support the volume of storage access that our proof procedure can be expected to generate.

5.8.5.1 Internal PE Structure

Figure 16 on page 109 shows the internal PE structure we propose.

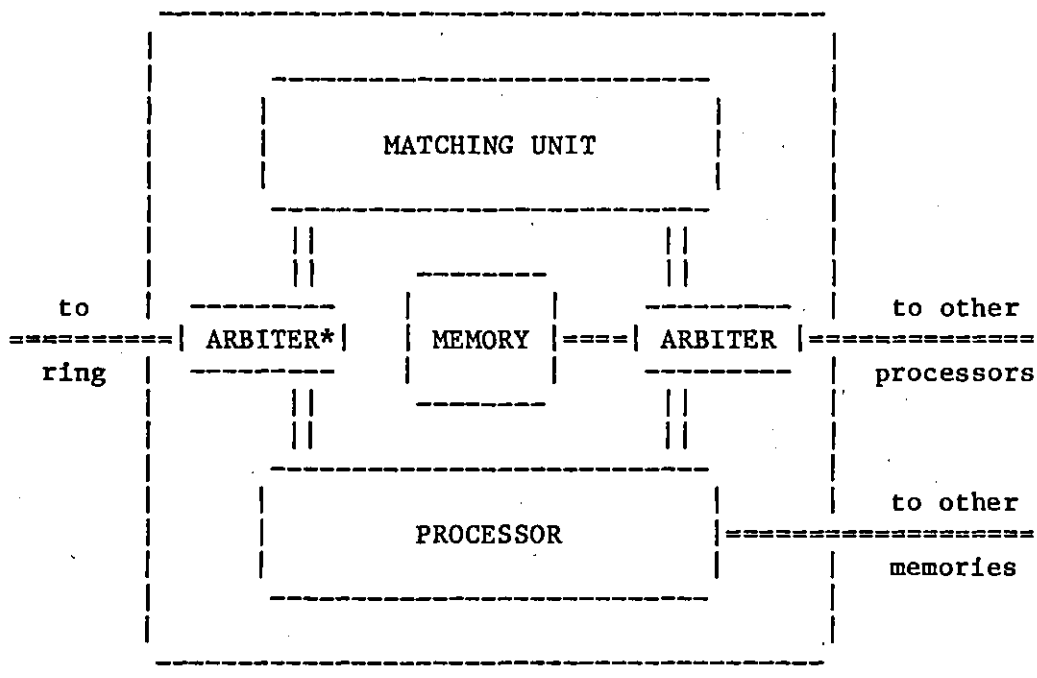


Figure 16.

The unit marked ARBITER* is the only part of the PE that sees the ring and arbitrates between the processor and matching unit for access to it.

Its connection to the processor serves to transmit packet-end signals and process-process messages from the processor and to receive packet-start signals after the processor has indicated that it is idle (indirectly, through a previous packet-end signal).

Its connection to the matching unit performs converse functions:- packet-end signals and process-process messages bound for the PE are recognised by that component and forwarded to the matching unit; packet-start signals, formed by the matching unit, are transmitted out via the ring.

5.8.6 Processor-Memory Connection

It is generally recognised that much research has been applied to the investigation of alternative processor-memory inter-connection strategies e.g. [34]. By and large, each proposal offers a trade-off between cost and average speed of access. To be able to decide on the best compromise calls for some investigation, both by evaluation and simulation, of the loading likely to be placed on the adopted scheme.

Below, we include three proposals aimed at lowering the number of accesses to external segments of memory.

1. Each PE will contain a local memory which includes a copy of the computer program that implements the proof procedure.

2. Each such local memory will also contain a copy of the user's Horn clause program. (This proposal will be modified in a later section.)

3. To organise the proof procedure in a way that exploits the direct link between a PE's processor and memory as fully as possible.

Item 1 above, by itself, makes at least half, probably significantly more - say 55-75% - of the accesses local. This is because in most processor architectures, the ratio of data accesses to instruction accesses is less than one-to-one, most such processors being built round the 0- and 1-operand instruction formats. (N.B. We are assuming that the proof procedure will be implemented by program code. Were it to be implemented in microcode, this percentage would be somewhat lower because fetching and executing the corresponding microcode would be faster operations. We are also assuming that local memories and segments of the global memory operate at the same speed.)

Item 2 also reduces the load significantly because structure-sharing ensures that the description of dynamic data is at least partly in terms of the static program. Hence any use of that dynamic data will in general involve local storage references. For

example, when matching two functors, the test which checks whether the two function symbols are the same will do so by making references to the static program stored locally in each PE. It is far more difficult to quantify the degree of network loading relief made possible by item 2 and extensive simulation would be needed.

We might make the guess that the first two items combined would make 70-85% of all storage accesses local to the PE making the access.

Item 3 seeks to fully exploit the link between a PE's processor and memory in order to reduce the loading on the (external) processor-memory network. One way in which this link is used is in the way that storage is managed and the next section will investigate this in some detail. Suffice it to say that both storage allocation and garbage-collection are undertaken by the processor in the PE whose segment of memory is the subject of the operation and hence such transactions may be done without burdening the processor-memory network.

We also point out here that ARBITER*, which interfaces the processor and matching unit to the ring, will be capable of diverting a newly formed packet-start signal straight to the attached processor, should the latter be idle. Because a matching unit is only concerned with processes based in its PE, it follows that the processor will then be dealing with a packet for a locally based process and this will tend to lower the demands on the processor-memory network, as was pointed out earlier.

Finally, we make the observation that if an implementation of the architecture consists of n PE's, then on average, a proportion 1/n of accesses between a processor and the segment of storage containing the data it requires, will just happen to be local accesses (the routing will, of course, be implemented in the ARBITER and will be transparent to the program). For large n, this contribution can be ignored, but may be considered significant for n less than 20.

Overall, we consider that items 1-3 above will result in no more than 10% of all storage accesses being external to the PE making the

access although we feel that the above analysis should be supplemented by appropriate simulation.

A small machine, perhaps of 10-20 PE's might have the processor-memory network implemented through a shared bus, the simplest option. Such an implementation might find favour in a small business machine or personal computer. A bus allows only one processor-memory access at a time and hence is a global resource. Although this violates our principle of full distribution, its implementation cost makes it an attractive proposition for such machines. For a more demanding application, a devolved interconnection strategy will be required and we suggest that the indirect binary n-cube network [35] or its generalisation the delta network [34] might be worthy of consideration. Our feeling is that the full crossover network is more powerful - and expensive - than the loading warrants.

For the shared bus implementation, the PE architecture may be further simplified by arranging for the storage ARBITER to also manage the PE's connection to the bus and this is depicted in Figure 17 on page 113. There is no loss in performance in doing this since the PE's processor and segment of global storage cannot both be active simultaneously via the bus.

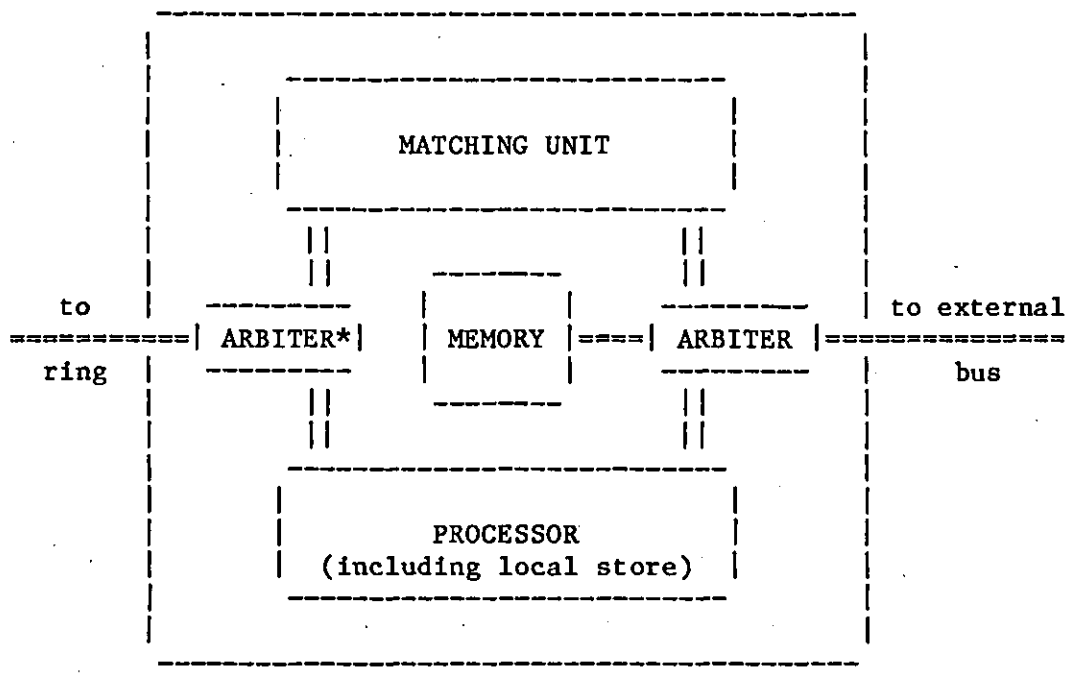


Figure 17.

We make the obvious observation that such an element seems a good candidate for fabrication in VLSI.

5.8.7 Storage Management.

We consider a machine consisting of n PE's to have its entire address space divided into n+1 parts. One part, copied in each PE for speed of access, holds the code implementing the proof procedure and also the user's program. The remaining n parts hold the n segments of PE memory. Thus a reference to an instruction of the inference system or to an expression in the user's program is identically understood in all PE's.

We propose that a PE's segment of storage be managed by the local processor. In particular, if a packet undergoing computation requires the allocation of storage - for instance, a packet recording a binding

in a register - the storage is taken from the pool of free store within the PE that accepted the packet.

If a PE runs out of free store, it discontinues the packet on which it was operating and turns its attention to garbage-collection. We will show presently, exactly what 'discontinuing' a packet involves.

In our proposal, storage is only released on the termination of processes and recovery of storage under these circumstances is amenable to implementation through a 'mark and scan' garbage-collection scheme.

In this scheme, each block of storage is marked 'no longer required' when this is determined to be the case. Garbage-collection need only take place when convenient and in our scheme, this would be whenever the processor determines that no more free store is available. At this stage, the entire segment of shared memory associated with the processor is scanned and marked blocks are returned to the pool of free storage. Notice that neither the allocation nor the garbage-collection of storage places a load on the external memory network - as was claimed earlier - (although marking, in general, does) and that by arranging for the memory ARBITER to give lowest priority to accesses from the associated processor, garbage collection will not materially affect the response to other demands made on the segment being tidied.

5.8.8 Process Control

We have indicated throughout this chapter that there exists an association between a process and "its" data structure and we will now formalise this association.

The execution of a process is organised around a process control record (pcr):-

pcr(messages, ref(data structure), packet state).

The first argument references the set of messages received by the process but not yet put into effect. In all probability, the set would be implemented as a queue of messages, the order reflecting the temporal sequence of message arrivals, and the first message in a non-empty queue would relate to the current packet for that process. This component is managed by the matching unit.

The second argument associates the process - as incarnated in the process control record - with its data structure. Notice the emphasis on the way this relationship is viewed here. For the purposes of controlling the computation, the data structure is regarded as merely an appendage of the process; it is processes and not data structures which are central to the implementation mechanism. ("ref(x)" names x through its location in store.)

The third argument is of significance only when the process is active - i.e. a packet for that process has been formed. It describes the processing that the packet requires by specifying the processor state that the receiving PE must establish once it accepts the packet-start signal. This information is supplied by the matching unit, which deduces it by examination of the message that gives rise to the packet. The information would include, for example, the program counter and settings for any relevant machine registers.

It can be seen that a packet is completely specified within the framework of its underlying process control record and so a packet-start signal may convey its information by naming the relevant process control record.

Termination of a process - and deletion of its associated data structure (e.g. termination of a main process after unification failure) - is effected simply by marking the storage in which the pcr and data structure reside and allowing garbage-collection to recover the store when convenient.

We are now able to show how a packet may be discontinued in one PE and restarted in another.

The crucial feature exploited here is that all PE's hold identical copies of the proof procedure and user's program and that all these copies appear in the same address space. Therefore, any location in local store holds identical contents in each PE and so any reference to an item in local store has the same meaning throughout the PE's. The same is trivially true of references to locations in global store. But the state of a processor, as explained above, is essentially the state of its program counter and other machine registers and these, by the above reasoning, will have the same significance to all processors and hence PE's.

Thus a mid-term packet transfer is effected by storing the mid-term state of the old processor in the packet state argument of the relevant pcr and releasing a packet-start message for it. The new PE will establish its processor state in the usual way from the third argument and will thus continue the processing of that packet.

In this way, our proposal solves the problem of a packet not being able to continue because the PE that accepted it has run out of store while others have not. It is seldom known in advance exactly how much store a particular process might need and any scheme which demands that storage be allocated from within the PE that first accepted the process runs the real risk of complications due to later insufficiency of free storage in that PE.

5.8.9 Modifications to the Basic Scheme

5.8.9.1 Specialist PE's

Earlier on, we said that under certain circumstances, it may be undesirable to hold a copy of the user's program in each PE and that specialist PE's, holding some procedure sets but not others may be a required feature. In the extreme case, the entire machine might be fully specialised, with no procedure set appearing in more than one

PE. We now indicate the modifications that need to be applied in order to cater for this requirement.

We generalise the requirement by applying specialisation to processes rather than procedure sets, that is, given a process, we will have the ability to execute its packets on some specified subset of the PE's. Thus we could arrange for all main processes concerned with a specific predicate symbol to execute on a given range of PE's or perhaps, if we so desired, we could arrange for the packets implementing a particular variable's registrar to be fully executed on a single PE.

The first change required is that start-packet signals will no longer be indiscriminately addressed but will instead be directed to the appropriate subset of PE's.

If only one PE can accept the packet, the packet-start signal may be directly addressed to it.

If more than one PE can accept it then the method described for ZMOB [36] is applicable. Here, the packet-start signal is addressed to the relevant subset of PE's by means of a capability code. Each PE also has a capability code of its own and the idea is that if a PE's capability code matches that of the message it sees on the ring then that message is accepted. The capability code of each PE must be set to reflect the processes that it can handle.

Direct addressing, capability code addressing and universal addressing can all be accommodated by the ZMOB ring.

Unfortunately, however, certain complications arise as a consequence of the PE's no longer being equal computational resources. The local overloading of PE's is one. Another is that of a PE running out of storage (after garbage-collecting all it can) - for if the same is true of all PE's capable of processing the packet that caused store to run out, computation on behalf of that packet cannot proceed. The smaller the subset of PE's able to accept such packets, the more likely this eventuality becomes.

Moreover, there are complications in the way that such a packet might be continued in another suitable PE, for the scheme we proposed made the assumption that the local address space of all PE's is identical. There are ways of overcoming this last difficulty (for instance by partitioning the PE's through an equivalence relation on the processes they can handle) but we do not wish to explore this further here.

5.8.9.2 Search Engine

An alternative method of avoiding repetition of large (ground) relations in all PE's is to hold them external to the multi-processor and access them through a search engine in the manner described earlier. This is the approach we favour, for we envisage a (single) parallel search engine - i.e. one able to respond to more than one query at a time - connected to all PE's. In this way, each copy of the user's program is identical (and will include copies of the necessary 'Consult Search Engine' clauses) and so an entirely distributed system, based on equal computational resources and allowing for large ground relations, is provided.

5.8.10 Regulation of Parallelism

The previous chapter made it clear that curtailment of parallelism was an important function of the proof procedure. The perceived activity of the machine should be used to decide when the degree of concurrency needs constraint. We offer the following mechanism for determining how and when parallelism should be restrained.

We have already indicated one way of adjusting to perceived machine activity - that of alternating between binary and n-ary branch naming. Perhaps a more satisfactory method of curtailing the degree of concurrency is by slowing down the distribution of packets that might be

expected to extend the search tree. These are packets beginning new main processes and follow from 'begin computation' messages from the parent main process. In this way, priority is given to the completion of processing for existing nodes. When the level of activity subsides once more, these packets may be distributed as normal.

Because any matching unit is able to inspect the messages it receives, it is able to identify 'begin computation' messages. Given a mechanism that makes the matching unit in every PE aware of overall machine activity, we can see that the matching unit is in a position to decide whether to locally buffer packets corresponding to such messages or to release them onto the ring.

Determination of overall machine activity should be done in a distributed manner and the means we propose involves each ARBITER* (which interfaces its PE to the ring) in monitoring the activity of the ring. A busy ring implies much external activity and we make use of this by arranging for the ARBITER* to communicate the level of activity to its attached matching unit. The matching unit then acts on this advice as described previously. (The same information may be conveyed to the processor, thus enabling it to switch between binary and n-ary branch naming as appropriate.)

This regulation of concurrency may be refined by noticing that a breadth-first exploration of the search tree is generally more concurrent than a depth-first one. By associating a tree depth level with each activation record, it is possible to constrain the degree of parallelism by giving priority to the 'deeper' main processes if the machine is adequately loaded. Thus matching units could order their buffers of waiting 'begin computation' packets in a way that will finely tune machine activity.

5.9 ASSESSMENT

5.9.1 Level of Parallelism

It is important to recognise that the principal criterion for judging the effectiveness of any proposal for concurrent computation is some measure of the level of useful concurrent computation and not the time it takes for a particular process to complete. After all, if every PE is fully involved in useful work, no improvement is possible and in particular, delays in the transmission of messages are of no consequence.

However, the criterion of having all PE's fully employed on useful work has to be satisfied in order to substantiate this view. In practice, the degree of achievable parallelism rather depends on the nature of the problem being solved as well as on the proof procedure itself (and might also depend on control advice).

It should be appreciated that a proof procedure organised around coarse grains of parallelism may not provide sufficient concurrency to satisfy certain machine/problem combinations, whereas one organised around finer grains may well do so.

At the other end of the spectrum, a design which cannot control the level of concurrency operating in the machine is liable to catastrophic failure if the nature of the problem presented to it gives rise to more parallelism than it can cope with - for instance, an unregulated Or-parallel scheme might run out of store because it is actively exploring too many branches simultaneously.

To characterise the behaviour of our design, we discuss two examples, each illustrating one of the above extremes and we show how our design copes with these situations.

5.9.2 Low Degree of Concurrency

In the Or-parallel proof procedure, the lowest degree of parallelism is exhibited by a deterministic refutation, one in which no more than one clause head unifies successfully with the selected goal. It it not particularly instructive to consider the case where the program only provides a single clause for each relation and we will assume that more than one clause may be invoked, in general, in response to a selected goal.

Typical of such examples is the deterministic use of a procedure set which consists of a base clause and a recursive clause, e.g. solving an Append goal in which two arguments are fully instantiated lists and the third is a variable.

Our scheme attempts to concurrently perform unifications between the selected goal and the heads of applicable clauses. In this respect, the Or-parallel scheme gains on the conventional backtracking one, which performs the unifications sequentially. Conery & Kibler's scheme is also organised around such sequentiality. In both cases, no progress in the refutation can be made while the clause with matching head is being sought.

However, our scheme, following unification, has to wait for bindings to be registered and this is an overhead from which the backtracking scheme does not suffer (Conery and Kibler apply substitutions explicitly). Moreover, in general, registration cannot proceed until a branch name has been allocated. In the earlier text, we indicated that for such deterministic refutations, pre-allocation of branch names is possible and we assume that this is done here. Thus the only delay we need to consider is that due to registration.

Because registration of bindings is concurrent, some communication and synchronisation via messages is required and this gives rise, as always, to timing delays in individual processes. These delays may be significant in our example because the PE's are not fully occupied: in fact, the only activity going on apart from registration is application of the selection function and establishment of child main processes together with their activation records. If registration is complete before such establishment, the registration delay is not sig-

nificant; otherwise it is.

In contrast, a conventional implementation will, at the time corresponding to registration, apply the selection function and repeatedly (until a unification success or exhaustion of suitable clauses)

1. Prepare an activation record
2. Attempt unification

Given a machine constructed according to our architecture and a traditional one, the ratio of registration time to the time taken to execute the above cycle will depend (amongst other things) on the nature of the particular program being run.

For instance, to show the Or-parallel scheme in a bad light, one might consider the procedure set

```
Append(z.u, v, z.w) <- Append(u, v, w)
Append(NIL, x, x)
```

and the goal <-Append(1.2.....NIL, NIL, y). In this case, the backtracking scheme, using LRDF search, will choose the correct clause every time except the last and for a long first list, the proportion of successful first-time choices will be high. The Or-parallel scheme will register four bindings on each recursion, one of them being an output binding which will be devolved (y and variants of w).

To show the Or-parallel scheme in a better light, one might consider the re-ordered procedure set

```
Append(NIL, x, x)
Append(z.u, v, z.w) <- Append(u, v, w)
```

and the goal <-Append(y, NIL, 1.2.....NIL). In this second example, the conventional implementation will, on each recursion except the last, select the wrong clause and successfully unify the first two pairs of terms before detecting that the Append atoms cannot be unified. The longer such abortive processing takes in comparison to registration, the more favourable the Or-parallel implementation appears.

As regards the reading of bindings, the two schemes are comparable, for whereas the backtracking scheme determines a binding (or that the variable is unbound) through two references, the Or-parallel scheme determines, in the corresponding two references, a set of bindings - the register.

An empty set indicates that the variable is unbound.

The only alternative in the deterministic case is a singleton set, whose binding applies because the search tree has a single branch, and this is indicated by an empty branch extension in the binding. A further storage access and a zero length bit comparison are also required in this eventuality.

In both cases, looking up a binding in the Or-parallel proof procedure involves a trivial amount of extra work.

In summary, our scheme, when used in such unsuitable circumstances, would not be capable of exploiting the resources it has available to it. Furthermore, the very aspects of the scheme designed to cope with more general (and favourable) cases may well delay execution. In other words, the overheads introduced to allow for parallelism - for instance, registration and a message transfer system - may not always be compensated for by concurrent program execution, in which case, we are left with a net deficit. It is conjectured that the same conclusion applies to all concurrent execution strategies, for the same reason.

Nevertheless, the above examples and reasoning lead us to believe that our scheme will provide an adequate performance under such unfavourable circumstances. Whenever the application allows more concurrency, more main processes can be expected to occupy the PE's and hence more concurrently executing useful work will take place.

5.9.3 High Degree of Concurrency

Our example here is centred on a database of employees. The prob-

lem is to find all employees, w, managed (directly or transitively) by A and possessing property P(w). We assume that each determination of P(w) is a significant computation in terms of the quantity of resources demanded.

```
<- Manager(A, w) & P(w)

Manager(x, y)
  <- Works-for(x, y)

Manager(x, y)
  <- Works-for(x, z) &
     Manager(z, y)

Works-for(., .)
Works-for(., .)
......etc......

P(t) <- .....
.....etc.....
```

(We choose to treat the Works-for relation outside the Search Engine context so as not to needlessly complicate our description.)

If we assume that binary branch naming is taking place, solutions of a Works-for subgoal, regardless of how that subgoal was derived, give rise to new branches of the search tree and such branches are begun more or less at the same rate as solutions of the Works-for subgoal are found (as explained earlier in the chapter).

Thus one can envisage the overall activity of the system gradually building up as new workers are discovered for known managers. This build-up is detectable from activity in the ring. Switching to the n-ary branch naming scheme will serve to delay the growing of the tree and help to conserve resources: branch names will tend to be shorter. (The delay does not matter in this case because we are assuming that the PE's are busy for a high proportion of the time.)

We may also hold back the development of the tree by delaying all packets seeking to begin unifications. Releasing these packets in a 'depth-first' manner will give priority to the extension of some branches of the search tree over others: those main processes further down the tree will receive more resources and this effect will be perpetuated until terminal nodes are encountered. Emphasis will be given to the deepest derivations and the example shows that once a terminal node is encountered (and the solution extracted in case of successful termination), storage, in the form of activation records and registers relevant to the particular instance of the P(w) subgoal concerned, can be released.

CHAPTER 6: AND-OR PROOF PROCEDURE

6.1 INTRODUCTION

The And-or proof procedure, as its name implies, seeks to exploit both forms of parallelism implicit in logic programs. It is somewhat different in nature from other schemes; in particular, it is not based on the producer/consumer notion of computation and so does not give the network form of and-parallelism found, for example, in the proposal put forward by Clark and Gregory [5]. In due course, we will describe the behaviour exhibited by the And-or scheme and show how the basic proof procedure might be adapted to display network behaviour.

Because the organisation of the And-or proof procedure is so different, this chapter will be primarily concerned with the design of the abstract scheme. We believe that the proposal represents a new direction of research and inevitably certain aspects of it will require further investigation: there are some known weaknesses which will be pointed out when encountered.

We have not been able to devote a great deal of effort to investigating the computational complexity of key parts of the scheme and thereby estimating their efficiency. Nor have we been able to formally show the proof procedure's correctness and completeness. However important these areas of research might be, their investigation must necessarily come second to the discovery and investigation of the basic scheme, and that is what we report here.

Although there exists a relationship between the And-or and Or-parallel proof procedures, it is not a particularly helpful one from the point of presentation. In fact, the Or-parallel proof procedure is a degeneration of the And-or scheme and the two schemes are compatible to the extent that a partial degeneration towards the former one serves as a regulator of parallelism in the latter. However, in the initial part of this chapter, we will concentrate on the unbridled And-or scheme because we prefer the approach that derives

the final proof procedure by restricting the degree of parallelism, rather than the approach that extends the minimal (Or-parallel) scheme by introducing new aspects to it.

The And-or proof procedure is based on the and-or tree representation of problem reduction, which is now described.

6.2 AND-OR TREE

The and-or tree model of problem reduction is well documented in the literature e.g. [33], [28]. Kowalski points out the major weakness of this model, viz. that it does not show the relationship between subgoals which are connected through shared variables. We will show how the And-or proof procedure overcomes this difficulty.

In the portrayal of the and-or tree given here, there are two types of node: goal nodes and (clause) head nodes.

The children of a head node are goal nodes.

The children of a goal node are head nodes.

The arcs connecting a goal node to each of its children represent the unifications of the goal with the respective heads. Alternative clauses give rise to alternative unifications and so the arcs are connected by v (or) operators.

A head node and its children represent a clause (more strictly, a clause which has been applied in response to the parent goal). The arcs connecting the head node to each of its children represent the links between the consequent atom and the goals derived from the antecedent atoms. The antecedents are conjoined and so the arcs are connected by & (and) operators.

A goal node with no child nodes is a fail node.

A head node with no child nodes is a success node.

The and-or tree for the following set of Horn clauses is illustrated in Figure 18 on page 129.

```
<-GOAL(y, z)
GOAL(u, v) <- P(u) & Q(v) & R(u, v)

P(1)
P(2)

Q(1)
Q(2)

R(w, 3) <- .... & ....
R(x, x)
```

The or-arcs are depicted as double lines to suggest interconnection through unification and to break up the tree into a more readily assimilated form. Viewing the tree from top to bottom, single line arcs lead to goal nodes, double line arcs lead to head nodes.

Notice that the and-or tree has a single root node and that therefore the top-level goal has to be atomic (this may always be arranged by including an intermediate 'GOAL' procedure, as in the above example).

6.3 INTRODUCTORY EXAMPLE

As motivation for the And-or proof procedure, we describe a very simple computation, based on the above example. Terms which are defined later in the main text will still be used (they will appear in bold-face here) and it is hoped that in most cases, their meaning in the context of this trivial example will be more or less obvious. Where this is not the case, we will give some intuitive justification

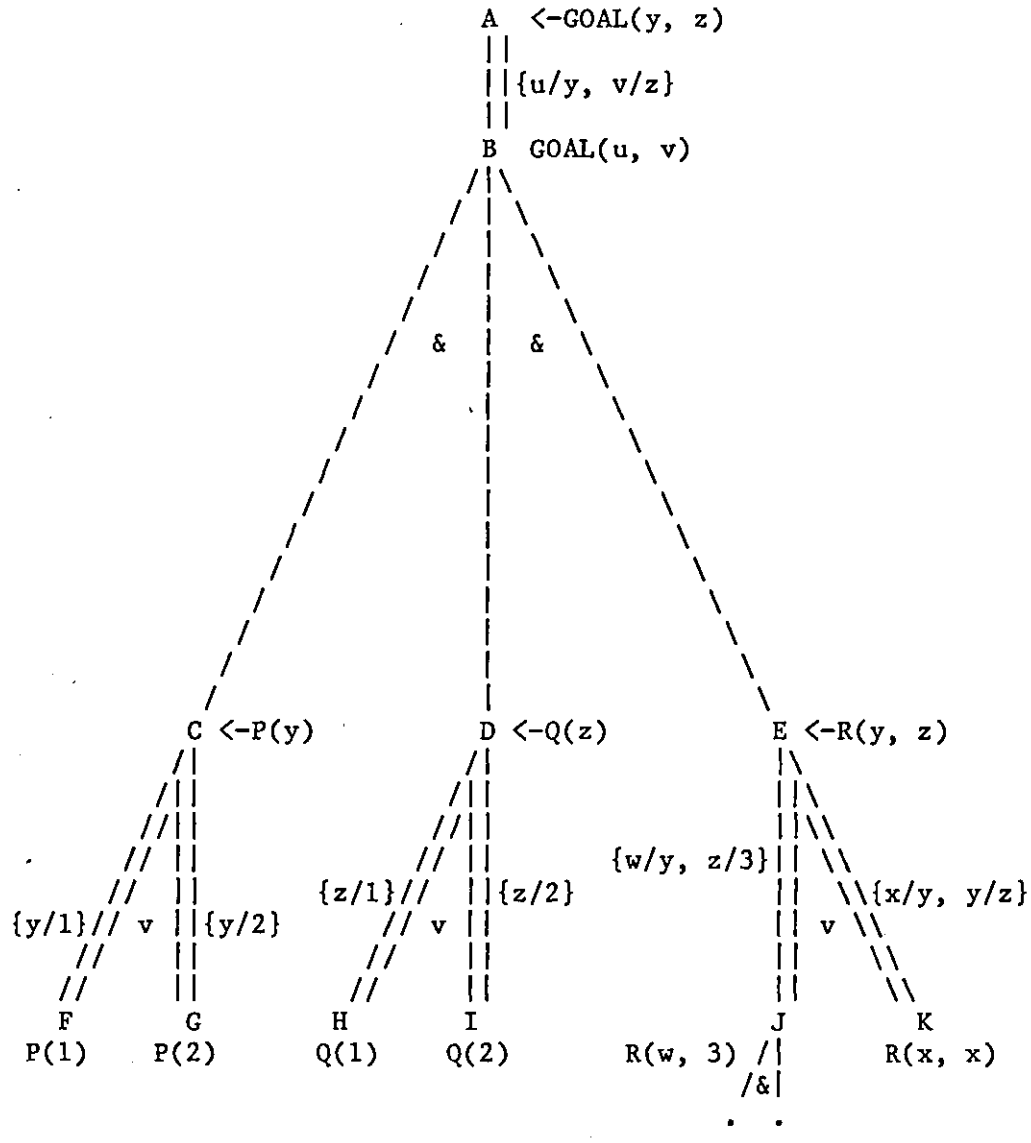


Figure 18.

to supplement their usage. In this way, the section may also be regarded as an overview of the scheme.

Description of Processing

Imagine the tree is grown breadth first. Suppose the point at which the nodes F to K appear has been reached.

The unifiers are summarised in Figure 19 on page 130 where, in these particular instances, the scope of a unifier is the singleton

set whose element is the name of the head node at which the unifier is established.

SCOPE	UNIFIER	SCOPE	UNIFIER
{B}	{u/y, v/z}	{F}	{y/1}
{G}	{y/2}	{H}	{z/1}
{I}	{z/2}	{J}	{w/y, z/3}
{K}	{x/y, y/z}		

Figure 19.

We may re-arrange the above information into the table of bindings shown in Figure 20.

VAR	BINDING+SCOPE	VAR	BINDING+SCOPE
u	y {B}	y	1 {F}
			2 {G}
v	z {B}		z {K}
w	y {J}	z	1 {H}
			2 {I}
x	y {K}		3 {J}

Figure 20.

(This re-arrangement of binding information reflects the storage scheme adopted in the proposal. However, it will become evident in due course that we still need the concept of 'unifier', that is, we need to associate bindings together in a manner that relates to the point at which they were made. In implementation terms, this means that any binding must be accessible not only through the variable name but also through the relevant unifier. Needless to say, any reasonable implementation will avoid a naive duplication of the binding information.)

Referring to the entries for z, the **scopes** of the various bindings for that variable are {H}, {I} and {J}. Scopes {H} and {I} are **disjoint** whereas scopes {H} and {J} & scopes {I} and {J} are **conjoint**.

Therefore, two **reconciliations** are set in progress with scopes {H, J} and {I, J} respectively. The first attempts to reconcile the components z/1, z/3; the second z/2, z/3.

They both fail.

The two failures result in **filters** with scopes {H, J} and {I, J} being established. The set of filters { {H, J}, {I, J} } is subsequently promoted to a filter with scope {B, J}, which is in turn **reduced** to the singleton {J} (since B is J's ancestor) and the branch leading to node J is **pruned**.

It may be useful to give the following justification for the steps in the above paragraph. The filter {H, J} formalises the 'incompatibility' between the head nodes, H and J, which appear as its elements. It states that there can be no solution, in the and-or tree interpretation of this term [33], which includes those two nodes. Similarly, the nodes I and J are incompatible in the same sense. Appealing to the structure of the and-or tree, we see that H and I represent the only ways to solve the goal at D and because the solution of this goal is a necessary requirement for use of the clause at B to be successful, we conclude that B and J are incompatible and state this fact by means of the filter {B, J}. However, since B is J's ancestor in the tree, it follows that any solution which involves J necessarily involves B anyway and so B's presence in the filter is

superfluous. Deletion of B from the filter leaves the singleton {J} and we interpret this as being sufficient grounds on which to discard that node with all its descendants and to abort all computations in that part of the tree.

Turning to the variable y, the scopes of its alternative bindings are {F}, {G} and {K}. Scopes {F} and {G} are disjoint but scopes {F} and {K} & scopes {G} and {K} are conjoint.

Consequently, two reconciliations are set in motion with scopes {F,K} and {G,K} respectively. The first one reconciles the components y/1 and y/z; the second the components y/2 and y/z.

They both succeed and return the unifiers {z/1} and {z/2} respectively.

At this stage, the set of unifiers is as in Figure 21.

SCOPE	UNIFIER	SCOPE	UNIFIER
{B}	{u/y, v/z}	{F}	{y/1}
{G}	{y/2}	{H}	{z/1}
{I}	{z/2}	{K}	{x/y, y/z}
{F,K}	{z/1}	{G,K}	{z/2}

Figure 21.

The bindings for the variables are as in Figure 22 on page 133.

VAR	BINDING+SCOPE	VAR	BINDING+SCOPE
u	y {B}	y	1 {F}
			2 {G}
v	z {B}		z {K}
x	y {K}	z	1 {H}
			2 {I}
			1 {F,K}
			2 {G,K}

Figure 22.

With reference to z in Figure 22, the scopes of the two new unifiers are tested for conjointness with the scopes of each of the previous unifiers binding z, and with each other. Five (separate) tests for conjointness are made:

{H} {F,K} CONJOINT, scope is {F,H,K}

{I} {F,K} CONJOINT, scope is {F,I,K}

{H} {G,K} CONJOINT, scope is {G,H,K}

{I} {G,K} CONJOINT, scope is {G,I,K}

{F,K} {G,K} DISJOINT

Component reconciliations are invoked in the first four cases

Terms: 1, 1 Result: Success, unifier is {}

Terms: 2, 1 Result: Failure

Terms: 1, 2 Result: Failure

Terms: 2, 2 Result: Success, unifier is {}

and the failures of cases 2 and 3 establish the filters {F,I,K} and {G,H,K} respectively. The sets of filters {{F, I, K}} and {{G, H, K}} are promoted to the filters {F, I, B} and {G, H, B} respectively. These latter filters are then reduced to the filters {F,I} and {G,H} respectively.

The unifiers current after these reconciliations have finished are depicted in Figure 23.

SCOPE	UNIFIER	SCOPE	UNIFIER
{B}	{u/y, v/z}	{F}	{y/1}
{G}	{y/2}	{H}	{z/1}
{I}	{z/2}	{K}	{x/y, y/z}
{F,K}	{z/1}	{G,K}	{z/2}
{F,H,K}	{}	{G,I,K}	{}

Figure 23.

Since no further reconciliations are indicated, the entire computation is complete and the solutions (i.e. candidate solutions not eliminated by the filters {F, I} and {G, H}) are

1. {B, F, H, K}

2. {B, G, I, K}.

The corresponding substitutions are

1. { {u/y, v/z}, {y/1}, {z/1}, {x/y, y/z}, {z/1}, {} }

2. { {u/y, v/z}, {y/2}, {z/2}, {x/y, y/z}, {z/2}, {} }

from which the sets of goal variable bindings, {y/1, z/1} and {y/2, z/2}, may be extracted.

6.4 THE BASIC SCHEME

Although the preceding example is a simple one, it does give the flavour of the And-or proof procedure. It should be clear from it that our scheme is driven by what we have termed 'reconciliation' and we now describe this central concept together with those to which it is closely related.

6.4.1 Reconciliation

Growing branches of the and-or tree independently leaves open the possibility that conflicting bindings will be made. Such conflict may come about if two or more unifiers, whose associated nodes of the tree contribute to the solution of different parts of the same problem, contain components with variable position occupied by the same variable.

The conventional description of a substitution stipulates that there can be no more than one component in the substitution for any given variable. This restriction is not, in fact, a necessary one since the concept of substitution is defined in terms of applying the substitution to an arbitrary expression: that is, replacing variables appearing in the expression by the terms to which those variables are bound in the substitution. The necessary restriction is that if more than one component for any variable exists in the substitution then it is immaterial which component is selected when the substitution is

applied to an arbitrary expression: the same instance of the expression (allowing for variants) is always computed.

We define a **substitution** to be a set of unifiers with the property that when applied - in the above sense - to an arbitrary expression, the resulting instance of that expression is independent of the choices made in selecting alternative components for the same variable. This unconventional meaning will be ascribed to the term 'substitution' throughout this chapter unless explicitly indicated otherwise.

As an example, the set of unifiers {{x/A}, {x/u}, {u/A}} is a substitution in the above sense because on application to an arbitrary expression, the end result is the same, regardless of which component for the variable x is chosen. Similarly, {{x/f(u,A), v/w}, {x/f(g(v),w), u/g(A)}, {v/A, w/A}} is a substitution whereas {{x/A}, {x/B}} is not.

Suppose S is a set of unifiers. If, for any variable v, S contains no more than one binding in which v occupies the variable position, then S is a substitution.

Alternatively, if S includes, within its unifiers, the two bindings v/t_1 and v/t_2 then it should be intuitively clear that if it is possible to unify t_1 and t_2 and to augment S by the unifier U so produced (giving S'), then it is immaterial which of the above two bindings is chosen when S' is applied to some expression involving v. We term this process **reconciliation** and, assuming the unification is successful, say that the bindings v/t_1 and v/t_2 are reconciled with **auxiliary unifier** U. If now it is possible to reconcile all pairs of alternative bindings stemming from S, including those introduced in auxiliary unifiers, then the resulting set of unifiers will be a substitution.

Reconciliation is essentially the 'unification of terms in unifiers', as Kowalski points out in chapter 4 of [28].

The following examples illustrate reconciliation

	S (Initially given)	U (Auxiliary)
1.	{{x/u}, {x/A}}	{{u/A}}
2.	{{x/f(u,A), v/w}, {x/f(g(v),w)}, {u/g(A)}}	{{v/A}, {w/A}}

For convenience, we will use the term **primary unifier** to refer to non-auxiliary unifiers. A primary unifier is associated with just one node of the and-or tree and is established during its growth.

6.4.2 Solutions

We will now describe what is meant by the term ´solution´ and in order to do so, we need the notion of ´candidate solution´.

Given a goal node G, a **candidate solution** of the goal at G is a set of head nodes, S(G), defined recursively as follows.

1. Choose one child (head) node H of G and include H in S(G).

2. For each child, G_1, G_2, ...G_n, of H, include $S(G_1)$, $S(G_2)$, ..., $S(G_n)$ in S(G)

3. Include no other nodes in S(G).

We will be primarily concerned with candidate solutions of the top-level goal and so unless otherwise qualified, the term ´candidate solution´ will relate to that goal.

A **solution** is a candidate solution whose associated unifiers have been reconciled. Consequently, the reconciled set of unifiers, including the auxiliary ones, is a substitution.

Notice that reconciliation is never indicated if subgoals are independent because in that case, no variables are shared between the

atomic goals and hence alternative bindings (for the same instance of each variable) cannot be made. In this event, every candidate solution will be a solution. In such circumstances, the And-or scheme shares individual solutions of the independent subgoals and fulfils the ideals described in Chapter 4 - whereby m+n (and not m*n) sub-computations are performed.

Example

Referring to the introductory example, consider the candidate solution {B, F, H, K} whose set of associated unifiers, S, is

{ {u/y, v/z}, {y/1}, {z/1}, {x/y, y/z} }.

For the candidate to be a solution, it is necessary to reconcile the components <y/1, y/z>.

The reconciliation succeeds with the unifier {z/1}, which is added to S. Inclusion of this unifier in S precipitates the need to reconcile the two bindings for z, viz. z/1 and z/1. This reconciliation succeeds with the empty unifier, which is also added to S.

No further reconciliations are indicated and so the candidate solution becomes a solution on addition of the auxiliary unifiers {z/1} and {} to S.

The above description regards the goal of transforming a candidate solution into a solution as the set of subgoals which seek to reconcile potentially conflicting components. Because reconciliations are independent one from another, we believe that there is no constraint on them being performed in any given sequence - or indeed in no sequence at all - and our proof procedure exploits this observation.

Before continuing the main exposition in the next section, we briefly digress to consider a simplistic approach to producing solutions. Our reason for doing so is to motivate the description of further aspects of the scheme.

A naive scheme for producing solutions might grow the tree to its limits, compute candidate solutions and then seek to reconcile alternative bindings in the candidates. We see two principal weaknesses in this approach.

1. It may happen that the and-or tree is not finite but it is known that nodes along a possibly infinite branch cannot contribute to a solution. In our example, nodes in the subtree rooted at node J cannot contribute to any solution but the naive scheme is unable to recognise this because it must first wait for the tree to be fully grown.

2. Much repetition is generally involved since the generation of candidate solutions is combinatorial in nature. Given two unifiers associated with head nodes N1 and N2, reconciliations of pairs of components (for the same variable) contained in each will be repeated for all candidate solutions which include N1 and N2. In particular, if N1 and N2 have many descendants, such repetition will be extensive.

6.4.3 Registration

As one might expect, one modification to the naive scheme is that component reconciliations will be performed eagerly, that is, as soon as alternative bindings for the same variable are detected. This will serve to overcome the first weakness.

We propose to conduct eager component reconciliations by carrying over, from the Or-parallel scheme, the notion of registrars. As before, a registrar exists for each variable and accepts alternative bindings for it. However, now it must take on the more active role of also initiating component reconciliations whenever these are indicated. We will shortly discuss the registration of bindings.

6.4.4 Scope

Registration by itself does not overcome the second weakness, that of repetition. What we would like to do is to share the results of component reconciliations between the relevant candidate solutions. To achieve this end, we introduce the notion of scope.

Each unifier is associated with a set of head nodes, called its scope. The scope of a unifier is a device that describes the substitutions within which the unifier is to appear. For the sake of readability, we will sometimes use the term "scope of a component" to mean "scope of the unifier in which the component appears".

The scope of a unifier is recursively defined as follows.

The scope of a primary unifier at the node N is the singleton {N}.

The scope of an auxiliary unifier, resulting from the reconciliation of components with scopes S_1 and S_2, is formed from the union of S_1 and S_2 by deleting nodes which are ancestors of others in the union.

The interpretation we place on the scope S of a unifier U is that if S is a subset of any particular solution then U is a member of the corresponding substitution.

Note that because a solution contains all ancestors of the tip nodes in that solution, it is safe to delete ancestral nodes after forming the union of S_1 with S_2: if S is a subset of some solution then that solution necessarily includes the deleted ancestral nodes anyway.

The notion of scope may be extended to the case where a component reconciliation fails. In this case, a filter, whose scope is computed in the same way as that for an auxiliary unifier, is produced.

The interpretation we place on a filter with scope S is that S is not a subset of any solution. Equivalently, no candidate solution which includes S as a subset is a solution.

Reference to the introductory example will serve to illustrate these concepts.

Note that a candidate solution need not necessarily be fully computed in order to determine whether a given scope is a subset of it. This observation allows us to abandon partially complete computations in appropriate circumstances.

6.4.5 Conjointness and Disjointness

A registrar exists for each variable and included amongst its tasks is the determination of whether or not two bindings submitted to it are for the same instance of the variable with which it is concerned - i.e. whether or not the two bindings might possibly relate to the same derivation. (An example of two bindings that do not relate to the same derivation is provided by the introductory example where the bindings concerned are z/1 and z/2, of scopes {H} and {I} respectively.)

Earlier, we showed how candidate solutions are formed and we indicated that within the unifiers associated with the candidate, all references to any given variable are references to the same instance of that variable. Consequently, reconciliations are relevant if and only if the scope of the resulting auxiliary unifier or filter is a subset of some candidate solution.

Below, we give a rule for determining whether or not a set of nodes is a subset of some candidate solution.

This rule is suggested by the following examples, based on the and-or tree extract shown in Figure 24 on page 142

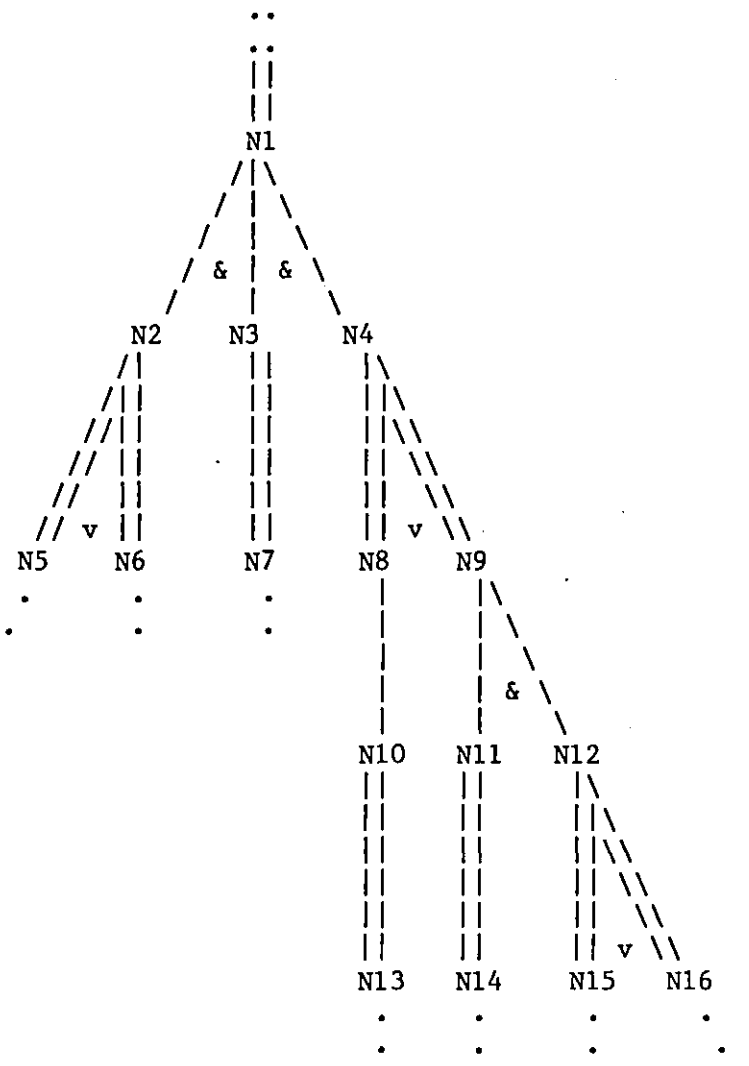


Figure 24.

1. The set of head nodes {N5, N7, N14, N15} is a subset of several candidate solutions.

2. The set of head nodes {N5, N7, N13, N15} is not a subset of any candidate solution.

The rule is that no two nodes in the submitted set of nodes - the supposed subset of some candidate solution - lie on branches concerned with solving some goal in alternative ways.

Equivalently, the rule is that for each pair of nodes in the set, their nearest common ancestor is a head node.

Two nodes are said to be **conjoint** if their nearest common ancestor is a head node.

Two nodes are said to be **disjoint** if their nearest common ancestor is a goal node.

In the second example above, the nodes N13 and N15 are disjoint and so the given set is not a subset of any candidate solution.

When presented with two components with scopes $S_1 = \{N_1, N_2, \ldots, N_n\}$ and $S_2 = \{M_1, M_2, \ldots, M_m\}$ respectively, it is not necessary to form the scope of the projected auxiliary unifier (or filter) and then test each pair of nodes for conjointness. It is readily verified that the n*m tests, with one node coming from each scope, will suffice. We will also say that two scopes are conjoint if their contained nodes are pairwise conjoint. The efficiency of the above tests will in practice depend on the chosen representation of scope and Kowalski's suggestion of a tree structure rooted in the nearest common ancestor would seem a promising line of investigation.

Example

As an example of the basic And-or scheme, consider the Shortlist program:-

```
Shortlist(in, n, out) <- Double(in, inter) &
                         Initial(n, inter, out)

Double(NIL, NIL)
Double(t.u, v.w) <- *(2, t, v) & Double(u, w)

Initial(0, s, NIL)
Initial(s(n), x.y, x.z) <- Initial(n, y, z)
```

supplemented by an appropriate definition of the '*' relation. The intended usage is for the user to supply a list of integers in the

first argument of the Shortlist goal, an integer in the second and a variable in the third. The computation instantiates this variable to a list whose length is equal to the second input argument and each of whose items is double the corresponding item in the first list. Thus

Shortlist(1.2.1.3.4.NIL, s(s(s(s(0)))), 2.4.2.6.NIL)

holds. For the purposes of this example, we will modify the program as follows:-

```
<- Shortlist(r)

Shortlist(p) <- Double(1.1.NIL, q) & Initial(s(0), q, p)

Double(NIL, NIL)
Double(t.u, v.w) <- *(2, t, v) & Double(u, w)

Initial(0, s, NIL)
Initial(s(n), x.y, x.z) <- Initial(n, y, z)

*(2, 1, 2)
```

The and-or tree is given in Figure 25 on page 145.

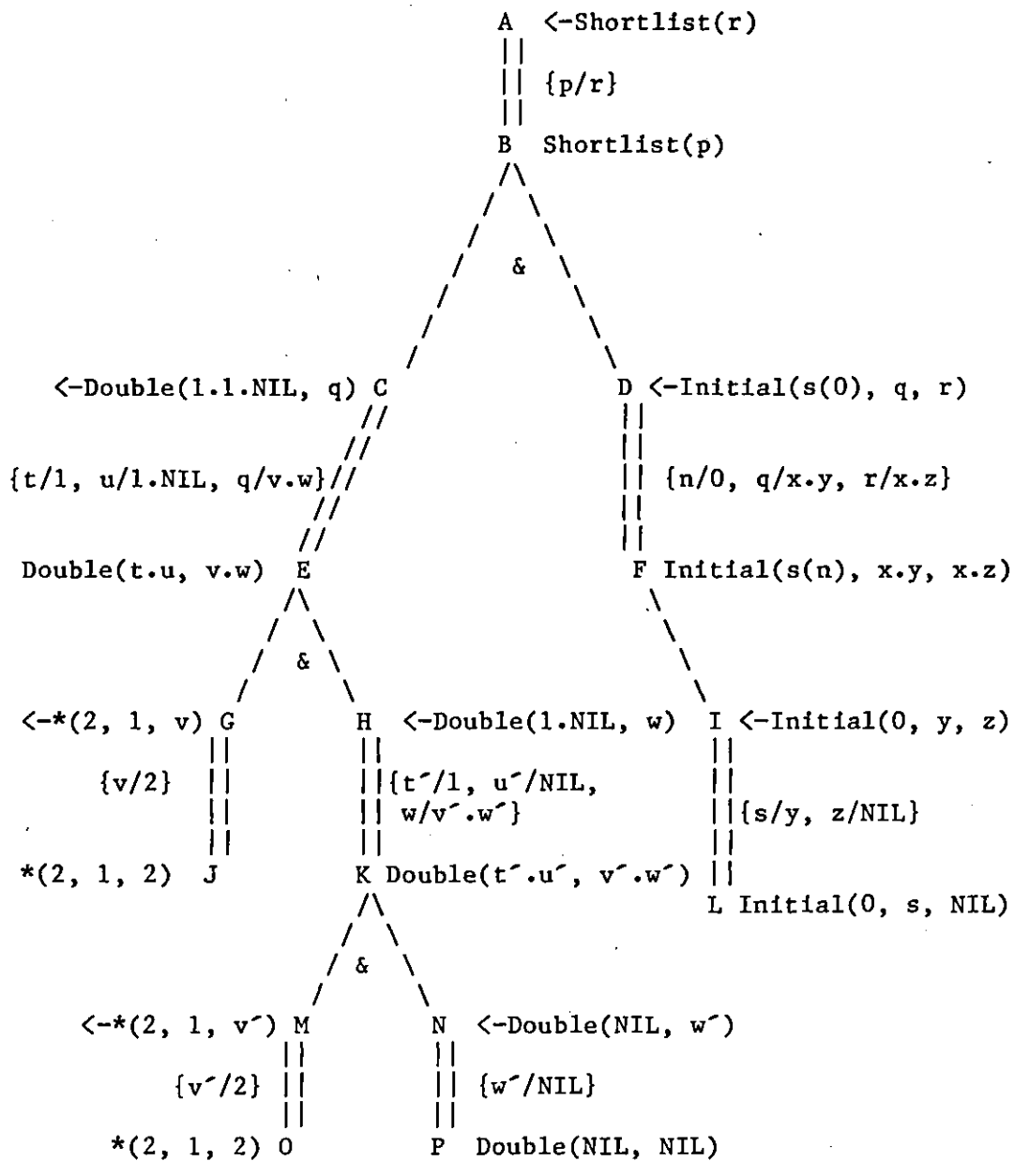


Figure 25.

We notice that the and-or tree confirms the ´functional´ nature of this example: all or-parallelism is very shallow and amounts to a failed unification in each case (not illustrated). Consequently, any test for conjointness will be satisfied.

It will be seen that the unifiers at nodes E and F have alternative bindings for q, viz. q/v.w and q/x.y. A reconciliation is set in motion and succeeds with auxiliary unifier {x/v, y/w}, of scope {E, F}.

It is easily verified that no further reconciliations are called for. Notice, however, that no solution is found until the entire computation terminates, for the existence of the intermediate list q needs to be established.

The solution is {B, E, F, J, K, L, O, P} with corresponding substitution comprising the set of unifiers at those above nodes, together with the auxiliary unifier {x/v, y/w}. From this substitution, one is able to determine the set of bindings relevant to the user's variable r, namely {r/x.z, x/v, v/2, z/NIL}.

6.4.6 Filtering and Pruning of the And-or Tree

We have described an and-or tree model of computation in which each node is tentatively assumed to be capable of contributing towards solutions when the tree is grown. As component reconciliations are attempted, failures occur and are described by appropriate filters. We now show how filters may be manipulated to (in general) curtail the growth of the tree.

The three operations introduced in this section - namely promotion, reduction and pruning - will be discussed with reference to the and-or tree extract shown in Figure 26 on page 147.

In this example, we assume the pre-existence of two filters with scopes {N9, N11} and {N12, N5}.

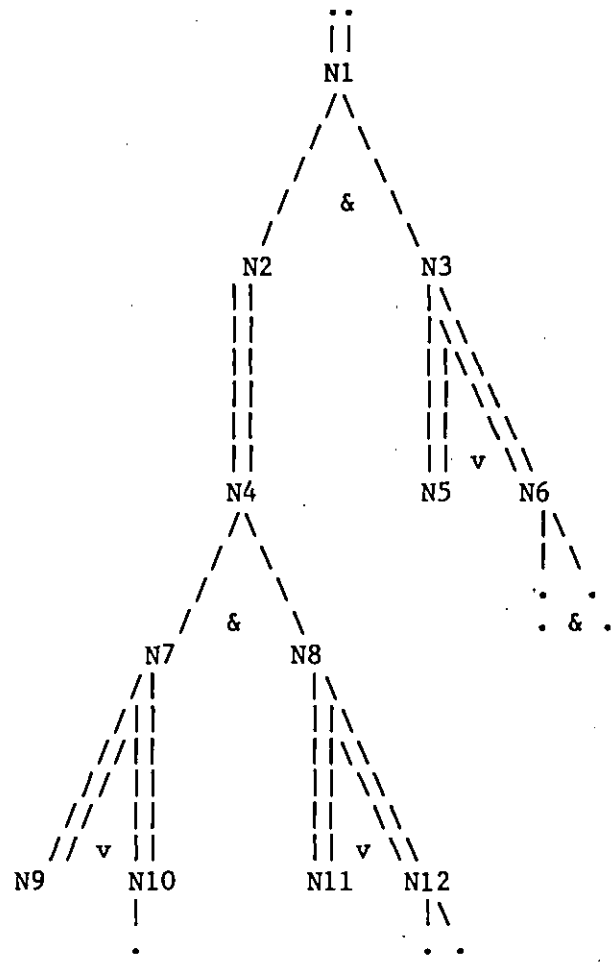


Figure 26.

6.4.6.1 Scope Subsumption

As a prelude to considering the manipulation of filters, it is helpful to introduce the notion of scope subsumption.

Scope S_1 **subsumes** scope S_2 if every node in S_1 is an ancestor of some node in S_2 ('ancestor', as always, is used inclusively).

For example (with reference to the above tree), S_1 = {N4} subsumes S_2 = {N10, N6}. When S_1 and S_2 relate to filters, the filter corresponding to S_2 is redundant:- for it states that no candidate solution which includes S_2 as a subset is a solution. This is already implied by the hypothesis that the filter S_1 subsumes S_2 (because all the ancestors of any node in a solution also appear in that solution).

6.4.6.2 Promotion

In this section, we consider a filter to exist if it is implied by some subsuming filter. The definition which follows is illustrated by Figure 27 on page 149.

A set of filters whose scopes are S_1, S_2, ..., S_n, may be promoted to form a filter with scope $S = S' \cup \{N\}$ where

S' is the intersection of S_1, S_2, ..., S_n and

$S_i - S'$ ($1 \leq i \leq n$) are singleton sets whose respective elements are all the children of some (goal) node N' and N is the parent of N'.

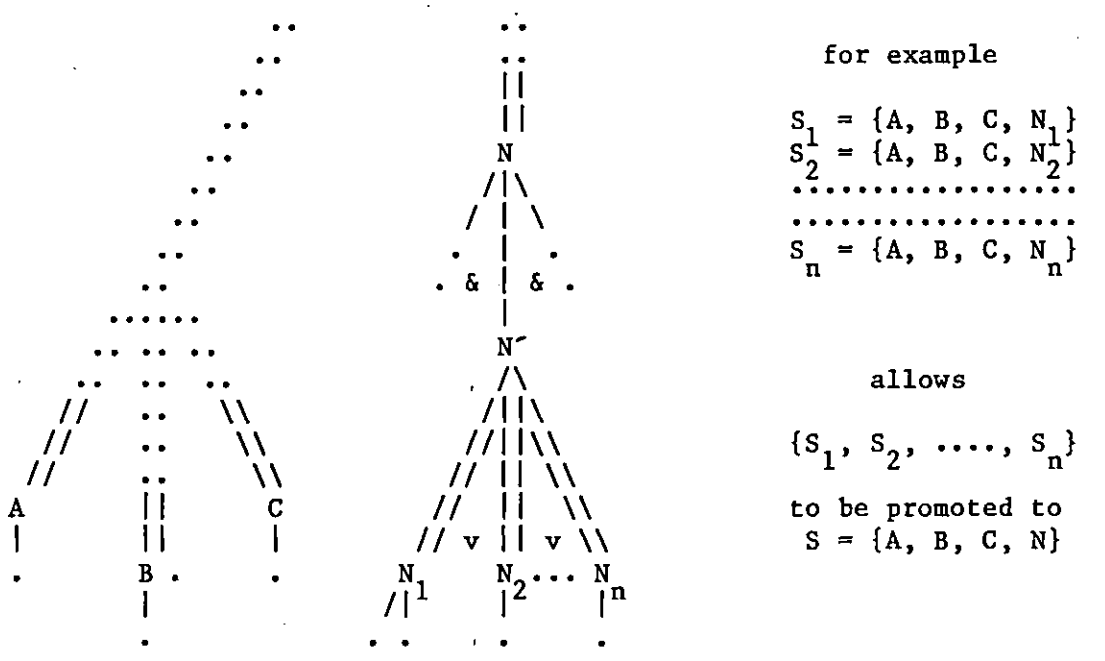


Figure 27.

Promotion of filters is justified by the following reasoning, where the names S, S', S_i, N and N' are used as above.

The filter, whose scope S_i is the union of S' with the singleton $\{N_i\}$ ($1 \leq i \leq n$) asserts that no candidate solution containing the nodes in S_i is a solution.

Any candidate solution that contains the grandfather node N must also contain a child node of N'. But if that candidate solution also includes all the nodes in S', a filter exists (its contents specified in one of S_1, S_2 , ... or S_n) to rule out the candidate.

We conclude that no solution can exist if it includes the nodes in S' and N and we signify this conclusion by establishing a filter with scope $S = S' \cup \{N\}$.

Since the scope S subsumes each of S_1, S_2, ... , S_n, it is seen that the new filter is equivalent to the previous set of filters.

As an illustration of promotion, suppose in our example that the filter {N10, N11} is produced and added to the pre-existing pair of filters, {N9, N11} and {N12, N5}.

Since N9 and N10 represent all ways to solve the goal at N7, it follows that the set of filters {{N10, N11}, {N9, N11}} may be promoted to the filter {N4, N11}.

6.4.6.3 Reduction

Promotion raises the possibility that the node which replaces its grandchildren is an ancestor of some other node(s) in the filter.

In this case, the filter may be made simpler by removal of the ancestral node. This process is termed **reduction** and is justified on the grounds that a candidate solution is precluded by the reduced filter if and only if it is precluded by the original one (since any candidate which contains the descendant node necessarily contains all of its ancestors, including the one deleted through reduction).

Continuing the above example, we see that because N4 is an ancestor of N11, it may be deleted from the filter, i.e. {N4, N11} may be replaced by the reduced filter {N11}.

6.4.6.4 Pruning

Deletion of a node from the scope of a filter raises the possibility that the resulting set is a singleton. Appealing to the definition of filter and the preservation of its semantics under promotion and reduction, the interpretation of a singleton scope is that the node concerned cannot be included in any solution.

Given a filter with singleton scope {N}, the rules of scope subsumption indicate that all unifiers and filters whose scope includes N or any of its descendants may be discarded. The significance of this is that the and-or tree may be modified by pruning the branch leading to node N.

In our example, the branch of the tree leading to N11 may be pruned.

Pruning, a computational notion, involves the following aspects:-

1. Deletion of unifiers and filters whose scopes are subsumed by {N}.

2. The curtailment of all computations whose projected unifiers or filters are subsumed by {N}. In particular, this includes computations which seek to extend the subtree descended from node N.

3. Indicating to N's parent that its child N has been deleted.

We regard pruning as purging the and-or tree of the specified branch, as though it had not been there in the first place.

If pruning a branch leaves the parent goal node without children, that node becomes a node of failure and pruning can then take place at the grandparent level. Thus in Figure 28 on page 152, pruning of the branch leading to node F allows the branch leading to node B to also be pruned, since there is no longer any way to solve the goal at C.

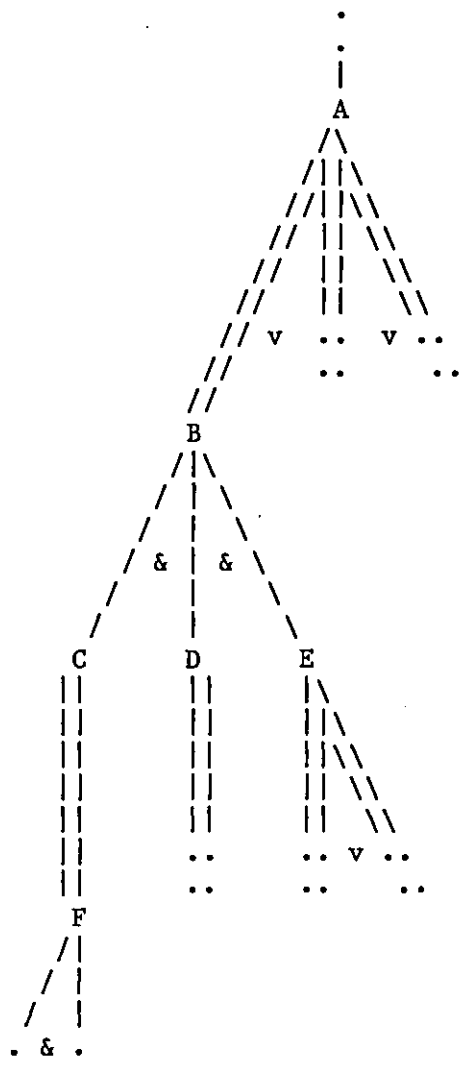


Figure 28.

The third aspect to pruning, that of informing a parent node that its child has been deleted, revokes a formerly indicated way to solve the goal at the parent node and raises the possibility that other promotions may thereby have become possible.

This is illustrated in our running example, where N12 is now the only route by which solution of the goal at N8 can be effected. The set of filters {{N12, N5}} may therefore be promoted to {N4, N5}. This may be followed by a promotion of {{N4, N5}} to {N1, N5}. N1 is an ancestor of N5 and so the filter {N1, N5} may be reduced to the

singleton {N5}, thereby allowing the branch leading to N5 to also be pruned.

Example

The manipulation of filters is well illustrated by the Same-leaves program, repeated from Chapter 4 in abbreviated form below.

```
S-1(a, b) <- L1(a, c) & L1(b, c)

L1( 1(d), d.NIL)
L1( e:f, g) <- L1(e, h) & L1(f, i) & App(h, i, g)

App(NIL, j, j)
App(k.l, m, k.n) <- App(l, m, n)
```

We keep the example simple by supposing that the given trees differ in their leftmost leaves and that these leaves originate from non-compound left branches - i.e. that the supplied goal statement is of the form

<-S-1(1(A):subtree-1, 1(B):subtree-2)

where subtrees-1 and -2 are arbitrarily complex ground terms.

Figure 29 on page 154 shows the and-or tree in simplified form. Lack of space obliges us to condense the right half of the diagram but this does not matter provided it is understood that the subtree descended from node F (concerned with the second term in the S-1 goal) directly corresponds to the subtree descended from node E (concerned with the first term).

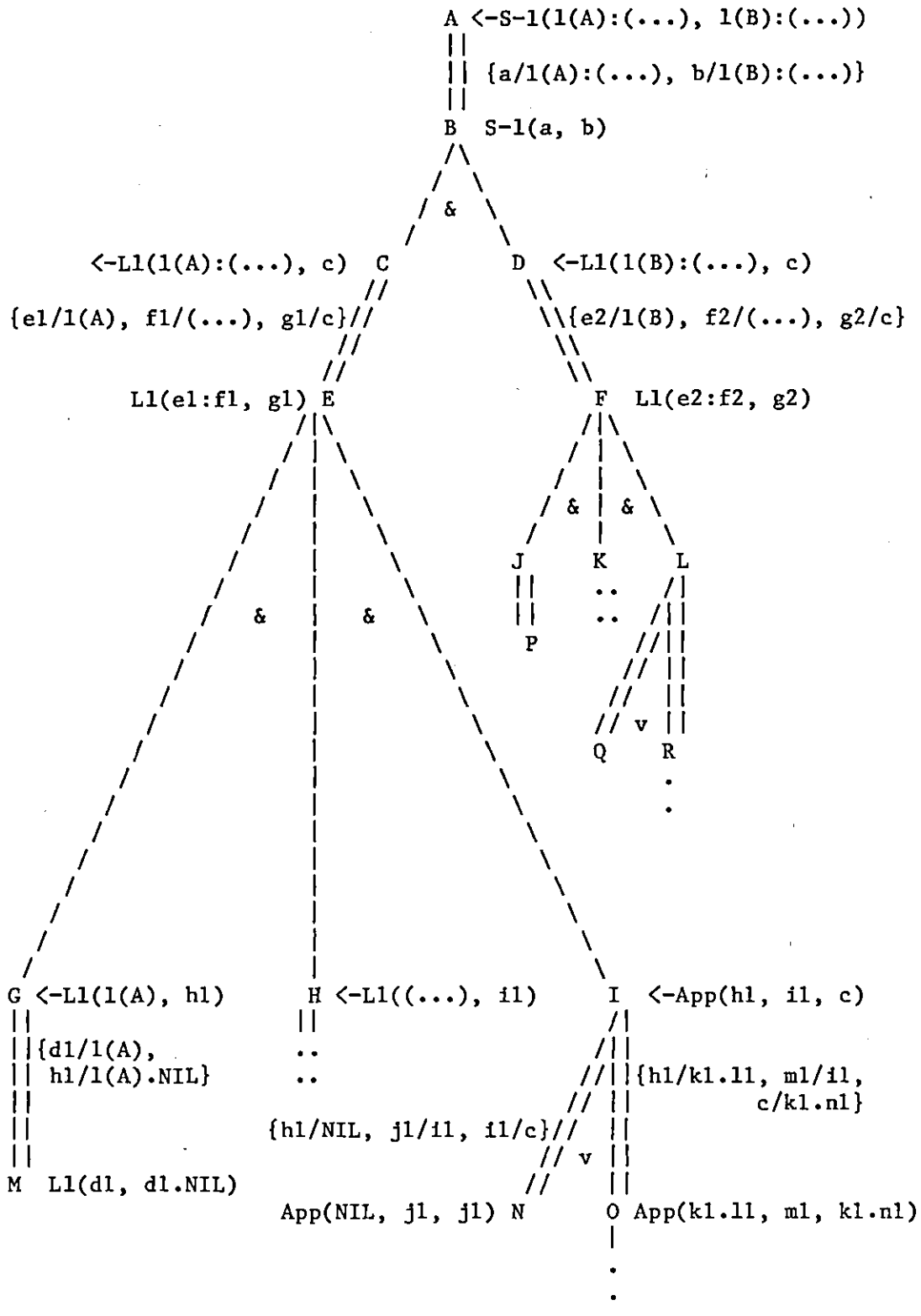

A <-S-1(1(A):(...), 1(B):(...))
{a/1(A):(...), b/1(B):(...)}
B S-1(a, b)
&
<-L1(1(A):(...), c) C
{e1/1(A), f1/(...), g1/c}
D <-L1(1(B):(...), c)
{e2/1(B), f2/(...), g2/c}
L1(e1:f1, g1) E
F L1(e2:f2, g2)
J
K
L
P
Q
R
G <-L1(1(A), h1)
{d1/1(A),
h1/1(A).NIL}
H <-L1((...), i1)
{h1/NIL, j1/i1, i1/c}
I <-App(h1, i1, c)
{h1/k1.l1, m1/i1,
c/k1.n1}
M L1(d1, d1.NIL)
App(NIL, j1, j1) N
O App(k1.l1, m1, k1.n1)

Figure 29.

We concentrate on those parts of the and-or tree expected to give rise to the failure, bearing in mind that other parts of the overall computation may be proceeding concurrently. We first turn our attention to bindings for the variable h1.

It will be seen from the and-or tree that h1 is bound as follows:-

h1/1(A).NIL	{M}
h1/NIL	{N}
h1/k1.11	{O}.

Scopes {N} and {O} are disjoint but each is conjoint with {M} and so reconciliations between the first and second and the first and third bindings are required.

The first reconciliation fails and this produces the filter {M, N}. The singleton set {{M, N}} of filters is promoted to {E, N} which in turn is reduced to {N}.

The branch leading to N is pruned.

The second reconciliation succeeds with auxiliary unifier {k1/1(A), 11/NIL}, whose scope is {M, O}. This binding for k1 will figure prominently in the eventual detection of failure.

Turning our attention to the other half of the and-or tree, we might expect that in the same way as the above binding k1/1(A) of scope {M, O} was made, the binding k2/1(B) with scope {P, R} will also, sooner or later, be made.

We now consider bindings for the variable c. It is readily seen that the bindings c/k1.n1 and c/k2.n2 for this variable (which represents the common leaflist) are contained in unifiers with scopes {O} and {R} respectively and that these alternative bindings will need to be reconciled. The resulting auxiliary unifier {k2/k1, n2/n1} will have scope {O, R}. There are now two bindings for k2:

k2/1(B)	{P, R}
k2/k1	{O, R}

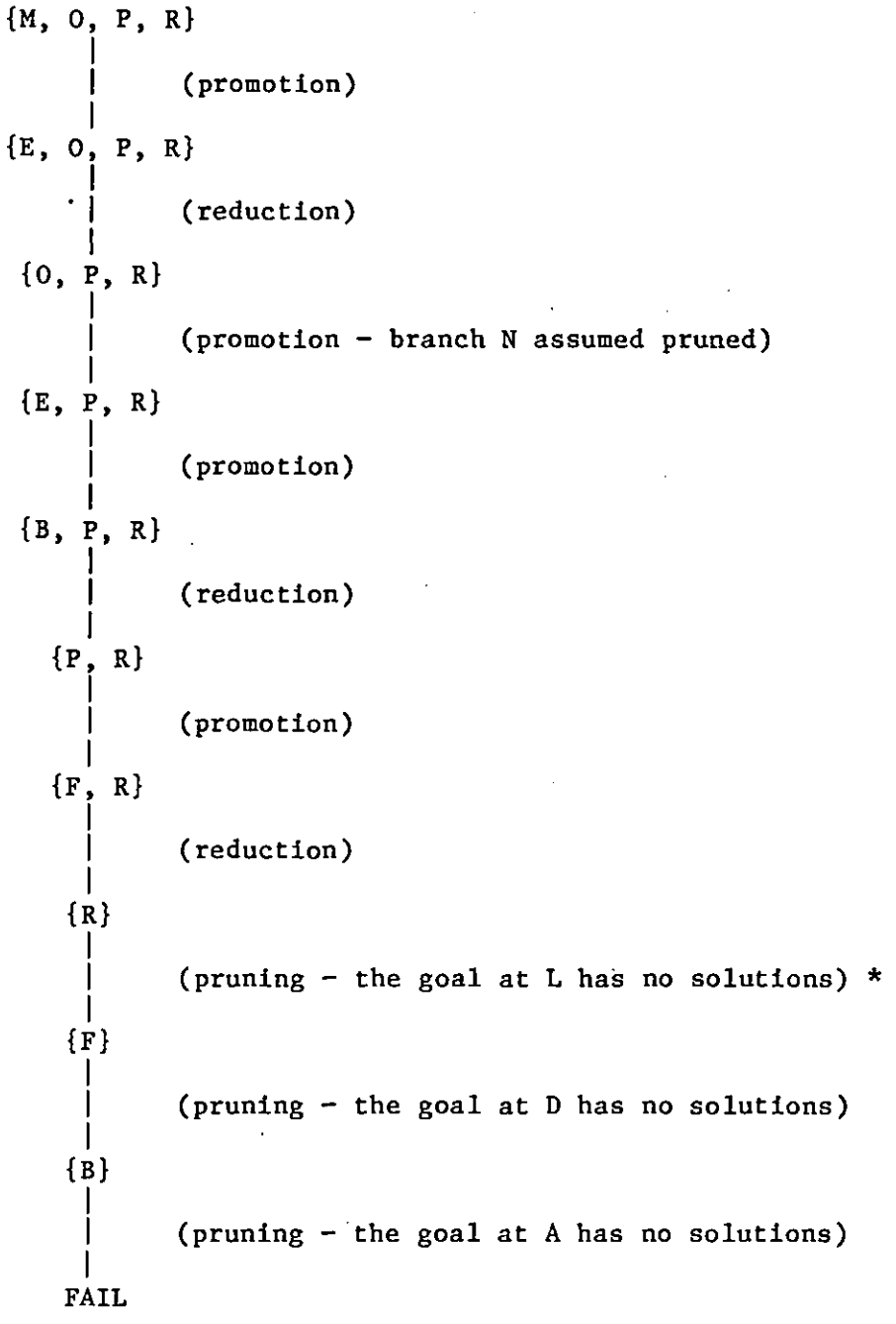
and these too will need to be reconciled since the respective scopes are conjoint. The resulting unifier will be {k1/1(B)}, whose scope is {O, P, R}.

We are almost done, for now we have two bindings for k1:

k1/1(A) {M, O}
k1/1(B) {O, P, R}

and since the scopes are conjoint, a reconciliation will be required - but this will fail and the filter {M, O, P, R} established.

Manipulation of this filter leads to the failure of the top-level goal, as is shown in Figure 30 on page 157.



* This step assumes that branch Q has already been pruned. If this is not the case then the arrival of filter {Q} will precipitate the failure indicated in the final steps above.

Figure 30.

Although the above example was based on one of many timing possibilities, it should at least be plausible from our earlier description that the same end result - that of failure at the highest level - comes about under any timing realisation. We will have more to say about timing considerations later.

6.5 CONTROLLING THE CONCURRENCY

In this section, we are concerned with ways in which the level of parallel activity might be controlled, both dynamically through the observation of machine activity and statically by means of program annotations. However, it is our belief that in the long term, all control must be carried out automatically since execution aspects such as this should be of no concern to the user.

6.5.1 Dynamic Control of Activity

One method by which parallelism may be regulated is based on the observation that extending the and-or tree tends to increase activity because it introduces, in general, more bindings for any given variable and may thereby promote the need for subsequent reconciliations.

On the other hand, reconciliations tend to have the opposite effect - for ultimately, they may cause branches of the and-or tree to be pruned.

Thus one means of control is to increase the level of activity by giving more priority to sub-computations that seek to extend the and-or tree (primary unifications) and decrease it by giving higher priority to sub-computations which seek to curtail it (reconciliations).

However, one needs to bear in mind that the above observation only refers to tendencies. It may well happen in particular cases that exploring all or-branches descended from a given goal node will turn out to be a profitable investment insofar as some consequential pruning will reduce later activity. One would need to investigate this area more fully to produce a balanced approach.

6.5.2 Suppression of Unproductive Parallelism

Consider the simple Grandparent program

Grandparent(x, y) <- Parent(x, z) & Parent(z, y)

and suppose that the goal statement <-Grandparent(u, v) is specified, that is, the user is interested in all pairs <u, v> in this relation. We will assume that the Parent relation is given exclusively in terms of ground assertions.

The basic And-or scheme, as described above, will break down the given goal into the two subgoals <-Parent(u, z) and <-Parent(z, v) and solve each independently. It will then reconcile alternative bindings for the shared variable z.

For a large extensionally held Parent relation, such a strategy is undesirable on two counts.

1. Most importantly, a gross amount of work is involved in growing branches for each assertion in the Parent relation (twice) and then setting up $O(n^2)$ filters, one per pair of non-reconciling bindings for z.

2. The whole point of allowing for and-parallelism in the first place is to permit the concurrent solution of conjoined subgoals. If a trivial amount of work is required to solve such subgoals, it is quite likely that the savings in elapsed time resulting from their

concurrent solution is far outweighed by the expense of organising the concurrency.

Sequential producer/consumer schemes do not suffer from these deficiencies and the first modification of our basic proposal is to allow concurrent and sequential execution to be mixed, in a manner we now describe.

6.5.2.1 Language Modification

In this modification, sequential execution is implemented through annotations, similar to those implementing sequences in [5].

Defined as part of the language are two conjunction operators, '//' and '&' whose declarative semantics are identical and the same as those normally given for '&'. Operationally, however, '//' and '&' respectively relate to concurrent and sequential execution.

In general, the antecedent of a clause is specified as a //-conjunction of &-conjoined atoms. For example, we allow an antecedent of the form

P1 & P2 // P3 // P4 & P5

('&' binds tighter than '//').

Following Clark and Gregory, we will term a string of &-conjoined atoms a sequence. Thus P1 & P2, P3 and P4 & P5 are three sequences.

The previously described And-or scheme relates to programs using the // conjunction throughout. As a first approximation to the operational behaviour of the new '&' conjunction, we will say that goals within a sequence are to be solved sequentially but that different sequences may execute concurrently with one other.

6.5.2.2 Modification to Proof Procedure

Suppose that the user's goal is given as

```
    <- GOAL
GOAL <- P1 & P2 // P3 // P4 & P5.
```

We require the and-or tree to display just three branches at the top, as depicted in Figure 31

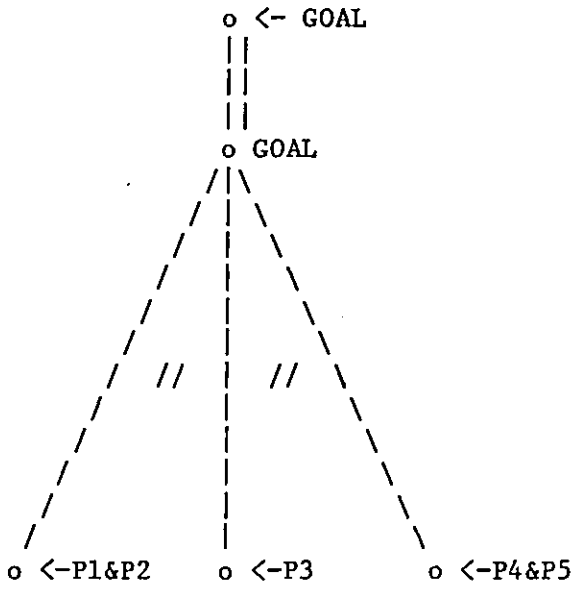


Figure 31.

It will be appreciated that, in general, goal nodes no longer relate to individual goals but rather to sequences of outstanding goals, exactly as one finds in the search tree model of computation. To this end, we need to assume the availability of a selection function for choosing the next goal from a sequence and, for the sake of simplicity, we will assume the familiar left-right, last-in-first-out rule. As before, the children of a goal node are head nodes, one for each clause whose head matches the selected goal.

The first approximation we gave above for the operational behaviour of the new operator '&' is not precise enough because it does not specify how existing goals - i.e. those not selected by the selection function - are to be passed down the and-or tree. There is some difficulty in giving a general rule and we see this area very much as one of future research. The difficulty arises in the following circumstances.

Suppose there exists a sequence at node N and that after selection of the next goal the depleted sequence S remains. If this goal is matched by the head of a clause whose body is

$$S_1 \;//\; S_2 \;//\; \ldots \;//\; S_n,$$

then the goals associated with nodes immediately descended from N are those found in S_1, S_2, ..., S_n and S. We would like the goals in S_1, S_2, ..., S_n to all be solved before the next goal in S is selected but this does not fit in well with our scheme. The weakness arises from the absence of any mechanism for synchronising the solution of conjoined goals and is related to another problem in the And-or scheme. We will return to this deficiency later in the chapter.

Example

The following example is based on the introductory example given at the beginning of this chapter and it may be found useful to re-examine that example now.

This time, the set of clause is specified as

```
<-GOAL(y, z)
GOAL(u, v) <- P(u) // Q(v) & R(u, v)

P(1)
P(2)

Q(1)
Q(2)

R(w, 3) <- .... & ....
R(x, x)
```

and the and-or tree is given in Figure 32 on page 164.

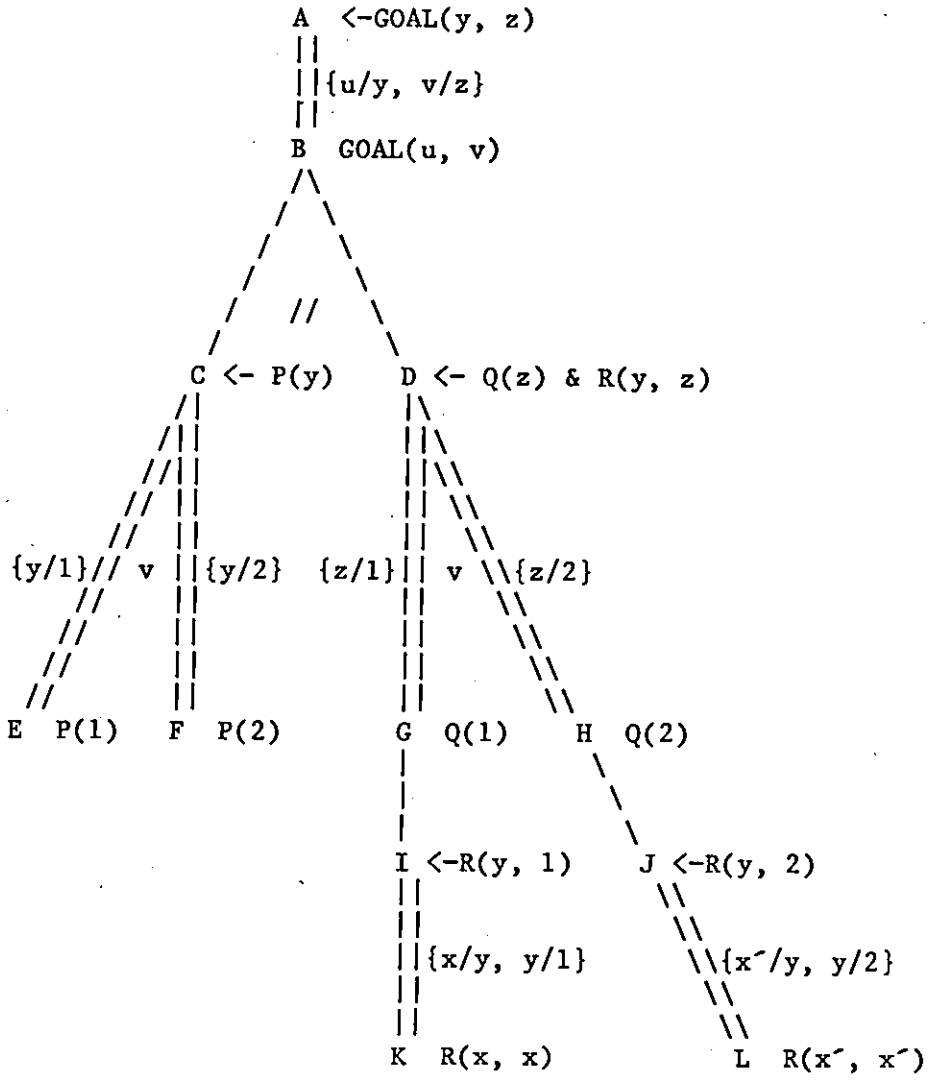


Figure 32.

Let us suppose that at some point in time, the unifications at nodes E, F, G and H are all completed. The unifiers are summarised in Figure 33 on page 165.

SCOPE	UNIFIER	SCOPE	UNIFIER
{B}	{u/y, v/z}	{E}	{y/1}
{F}	{y/2}	{G}	{z/1}
{H}	{z/2}		

Figure 33.

The bindings for the variables are as shown in Figure 34.

VAR	BINDING+SCOPE	VAR	BINDING+SCOPE
u	y {B}	y	1 {E} 2 {F}
v	z {B}		
z	1 {G} 2 {H}		

Figure 34.

The two bindings for y do not give rise to a reconciliation because the scopes {E} and {F} are disjoint. Scopes {G} and {H} are likewise disjoint and so reconciliation is not indicated for z's two bindings either.

Suppose that the tree is now grown to its limit. We note that unification of the goal <-R(y, 1) with head R(w, 3) fails and so node I only has one child node. Similarly for node J. At this stage the unifiers and bindings are as summarised in Figure 35 and Figure 36 respectively.

SCOPE	UNIFIER	SCOPE	UNIFIER
{B}	{u/y, v/z}	{E}	{y/1}
{F}	{y/2}	{G}	{z/1}
{H}	{z/2}	{K}	{x/y, y/1}
{L}	{x'/y, y/2}		

Figure 35.

VAR	BINDING+SCOPE	VAR	BINDING+SCOPE
u	y {B}	y	1 {E}
			2 {F}
			1 {K}
v	z {B}		2 {L}
		x	y {K}
z	1 {G}		
	2 {H}	x'	y {L}

Figure 36.

As far as the bindings for y are concerned, scopes {K} and {L} are disjoint but each of the pairs {E} & {K}, {E} & {L}, {F} & {K} and {F}

& {L} are conjoint and so the four corresponding reconciliations are set in motion. These result in filters {E, L} and {F, K} being established, both other reconciliations succeeding with empty auxiliary unifiers.

Since no further reconciliations are indicated, solutions may be derived by generating candidates and discarding those precluded by the filters {E, L} and {F, K}.

The resulting solutions are {B, E, G, K} and {B, F, H, L}.

Notice that the total amount of computation in this example is less than that for the earlier version. We cite this observation as support for the conjecture that by increasing the degree of parallelism in an algorithm, one generally increases the total workload.

6.5.2.3 Relationship to Or-parallel Proof Procedure

Here we are concerned with establishing the relationship between the modified And-or scheme and its Or-parallel counterpart. It is readily verified that in the extreme case when all conjuncts are of the sequential &-form, the resulting and-or tree is isomorphic to the corresponding search tree. In this event, each head node in the and-or tree has no more than one child goal node. The isomorphism merely coalesces every goal node (which now of course represents a sequence of outstanding goals rather than a single goal) with its parent. The isomorphism is illustrated in Figure 37 on page 168.

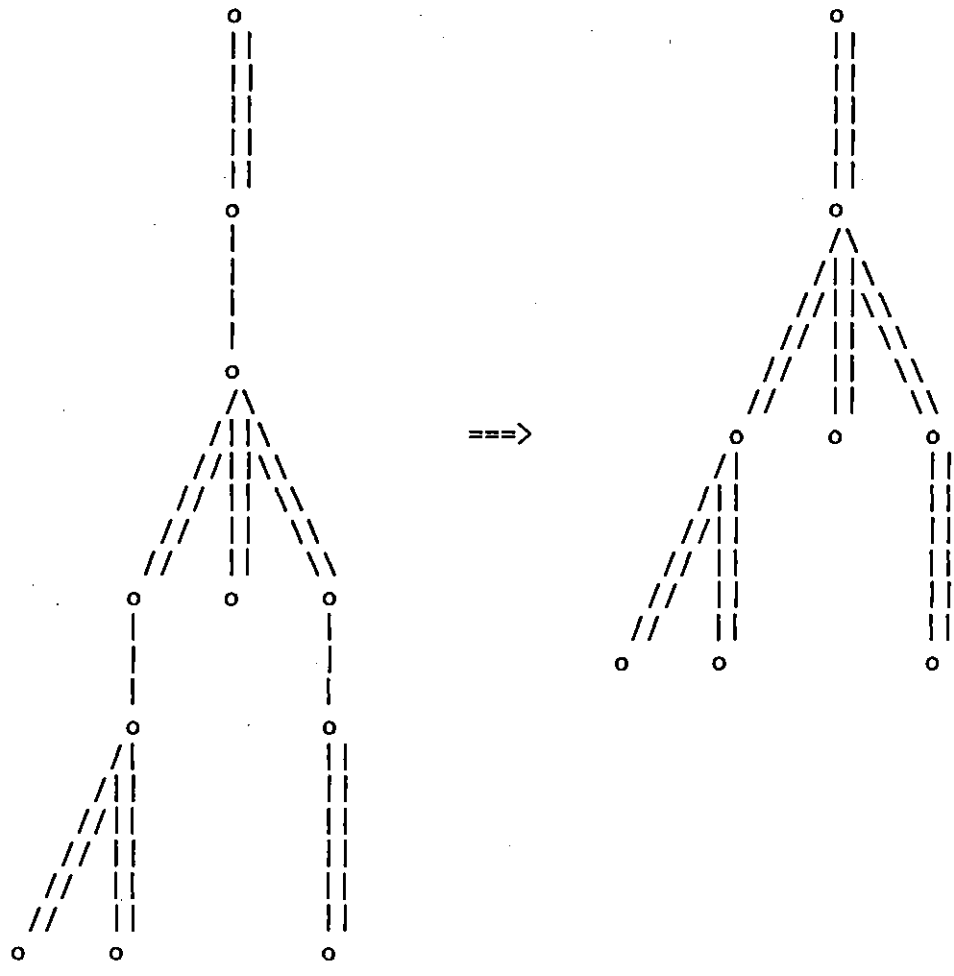


Figure 37.

More importantly, the And-or proof procedure essentially reduces to the Or-parallel proof procedure because reconciliations are never indicated: all nodes are pairwise disjoint.

6.5.3 Networks

The introduction of sequences into our scheme does not, in itself, give a fine enough control over the execution of programs. Essentially, the basic scheme gives us the maximum available and-parallelism but if this is tempered by the use of sequences, we may finish up by

losing more and-parallelism than we would wish. This is illustrated by the Compact example given in Chapter 4, which is repeated here.

```
Compact(NIL, NIL)
Compact(u.x, u.y) <- Remove(u, x, z) & Compact(z, y)

Remove(u, NIL, NIL)
Remove(u, u.w, w') <- Remove(u, w, w')
Remove(u, v.w, v.w') <- ¬(u = v) & Remove(u, w, w')

x = x
```

Remove, in the sequenced version of this program shown above, would delete all copies of the first item in the supplied input list to produce the intermediate list z. This list would be fully computed before Compact(z, y) is selected and there would be no and-parallelism.

Were the program to be modified so that the Remove and Compact atoms become conjoined by '//' rather than '&', too much and-parallelism would result since the Compact goal would attempt to generate all tuples in the corresponding relation - no account of Remove's output would be taken - and execution would run out of control.

The modification we propose below allows us to obtain execution behaviour between these two extremes.

As in [6], [5], we would allow the user to mark producers by means of annotations - e.g. in the above example we might mark Remove as the producer of z. (Alternatively, annotations might be generated by input-output mode declarations.)

The basic And-or scheme operates as previously described until such time as some unification U attempts to bind a variable - z say - for which it is not a producer. Up till this point, all bindings made by U will be equally valid for all possible bindings of z. These bindings are registered in the usual way, together with the distinguished binding z/?.

Other schemes, e.g. [5], which have no real or-parallelism, would leave the partially complete unification suspended and restart it when a suitable binding came to hand. Our scheme needs to terminate the partial unification and continue the residual matching at the point of interruption for each alternative binding of z. Since the outcomes of these continuations will, in general, depend on the particular binding assumed for z, we must be able to describe the context in which the unifiers or filters apply. We now indicate how this is done and as one might suppose, the method is based on the idea of scope.

Suppose then that U represents a unification which attempts to bind the variable z and that U is not a producer of z. Suppose further that S is the scope of the unifier being generated by U.

On attempting to bind z, U terminates with success and the bindings already made are registered with scope S. Let us denote these bindings by Unifier[root]. The binding z/? is also registered (scope S) and the register entry carries with it some indication of which pairs of terms remain to be matched.

Suppose now that a normal binding z/t is produced. Assuming that S and the scope of this new binding are conjoint - i.e. assuming that both bindings z/? and z/t refer to the same instance of z - we 'reconcile' these two bindings by establishing a unification, U[t], that seeks to match the pairs of terms indicated as part of the register entry for z/? and which reads the binding z/t for z. Because this continuation is dependant on the particular binding z/t read, we signify this fact by ascribing to U[t] the scope S[t] formed in the usual way from S and the scope of z/t.

The above description extends the definition of reconciliation to these circumstances.

Should an alternative binding z/t' be produced, a second unification continuation U[t'] comes into being under scope S[t'] and this runs independently of the first etc..

It should be evident that this application of scope is entirely consistent with its previous usage: for example, if U[t] fails its

unification then a filter with scope S[t] will result and this will rule out corresponding candidate solutions; it will not, however, necessarily rule out candidates which relate to the binding z/t' since S[t] will not be a subset of those candidates.

Suppose now that S[t] is a subset of some solution. Because S (the scope of the root unifier) subsumes S[t] - the latter scope was computed from the former - it follows that the root unifier will appear in the corresponding substitution. In this way, the root unifier is shared amongst all relevant substitutions. Conceptually, this sharing may be visualised in another way. If Unifier[t_1], Unifier[t_2], ..., Unifier[t_n] represent the outputs of the successful continuations then we may consider them as alternative extensions of the root unifier. If now some of these continuations themselves are subject to similar interruptions on account of other variables then the initial unification could be viewed as a tree structure of alternative continuations (Figure 38 below).

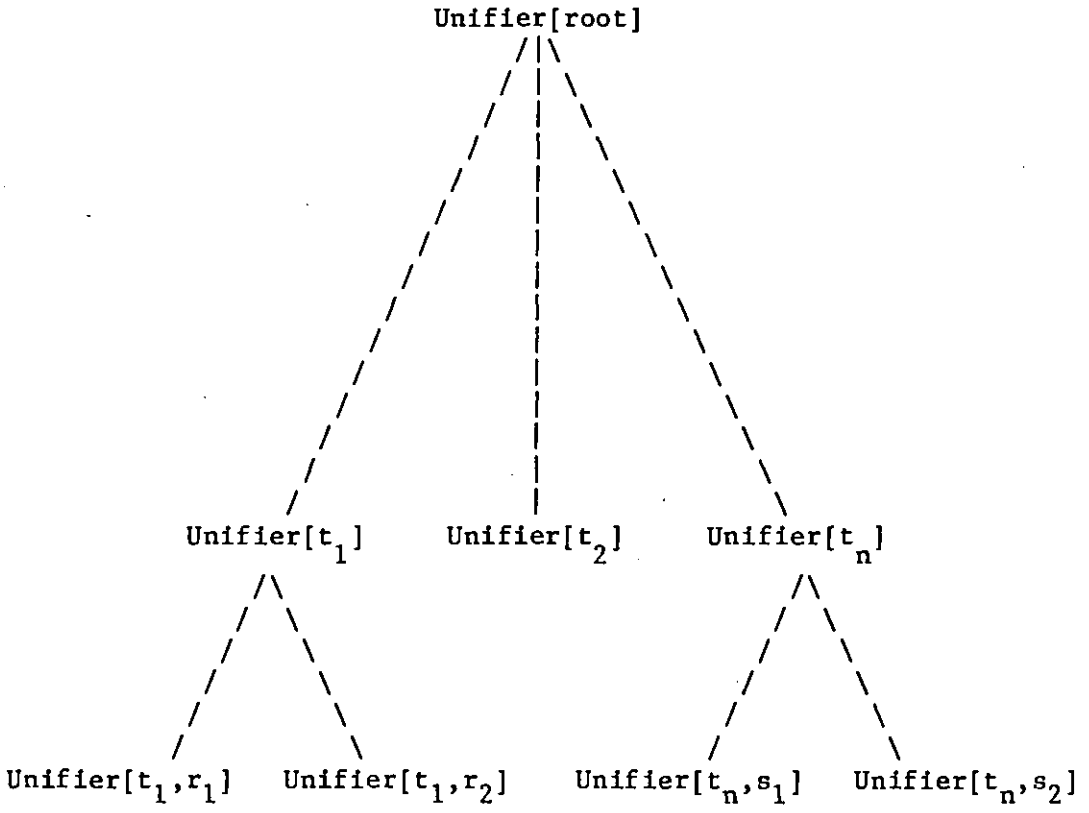


Figure 38.

The above means of control is more general than that given earlier and might be used throughout in place of the former. However, the exact behaviour would not then be the same, although both modifications, if used appropriately, will have a moderating effect. An evaluation of the two modifications would be a useful exercise and might indicate that for practical purposes, the first is superfluous.

6.6 IMPLEMENTATION CONSIDERATIONS

As indicated earlier, we do not intend to give an implementation design here but will content ourselves with indicating how certain aspects of an implementation might be effected.

We have tried to make this section as comprehensible as possible but some parts of it are very detailed. Our motives here are to impart as complete a picture of our scheme as possible and some of the difficulties we foresee can only be appreciated after a relatively full description.

6.6.1 Structure-sharing

As one might suppose, the same reasons that led us to choose structure-sharing principles for the implementation of the Or-parallel proof procedure lead us to choose them again here.

A principal feature of the And-or scheme is its organisation around the idea of sharing unifiers among as many substitutions as possible. Thus the reasons for wanting structure-sharing are even more compelling here than they were for the Or-parallel scheme, where, in turn, they were stronger than for the conventional backtracking implementation. (One might speculate that this is a consequence of more parallelism in the proof procedure but we will not pursue this conjecture any further.)

6.6.2 Unification

The design we have in mind is based on conventional ideas of representing expressions by means of structure-sharing. Bindings would not be applied explicitly but would be looked up whenever needed, exactly as in the Or-parallel proof procedure.

We carry over the idea of registering bindings but now each register entry takes the form <term, scope>, rather than the simpler <term, branch> of the earlier scheme.

For primary unification, that is, unification associated with a single node of the and-or tree, the rule for looking up a binding in a register is essentially the same as it was in the Or-parallel proof procedure: the sought binding must have been made somewhere along the node's ancestral branch. The scope of such a binding will be a singleton whose element names a node which is an ancestor of the node at which primary unification is being undertaken.

For an auxiliary unification, this rule has to be modified.

Suppose a reconciliation is indicated between two bindings and suppose the bindings concerned originate from unifiers with scopes S_1 and S_2. The scope, S, of the auxiliary unifier (or filter) resulting from the reconciliation (attempt) is computed according to the rule given earlier.

If, in the course of the auxiliary unification, a binding for some variable v has to be looked up, then any binding for v, whose scope S' subsumes S, will do: the final result of the overall computation, the substitutions, will each have the same effect when applied to an arbitrary expression, no matter which binding for v is chosen.

Such a binding is called a **candidate binding** for the evaluation of v in the context S.

The justification behind this rule, remembering that S' subsumes S iff all the nodes in S' are ancestors of nodes in S, is that if S is a

subset of some solution then so is S´. Hence all candidate bindings for v will appear in the corresponding substitution.

Note that the rule given above for primary unification is a special case of this more general rule.

6.6.2.1 Choice of Bindings and Timing Considerations

The notion of candidate bindings raises the prima facie complication of possibly having more than one candidate for a given evaluation.

Let us suppose that some unification is being attempted and that the scope of the resulting unifier or filter is S. Suppose further that the variable v is to be evaluated in the course of unification and that candidate bindings

$$c_1: \langle t_1, S_1 \rangle \text{ and } c_2: \langle t_2, S_2 \rangle$$

exist for that variable.

It may be helpful to consider an example when following this argument and we provide one which refers to Figure 24 on page 142:

S: {N5, N7, N14, N15}
c_1: <x.y, {N5, N7, N9}>
c_2: <A.NIL, {N5, N15}>.

Since c_1 and c_2 are candidate bindings, S_1 subsumes S; likewise, S_2 subsumes S.

We note that the nodes in S_1 are pairwise conjoint with respect to those in S_2, for all nodes are ancestors of those in S. (If node N_1 in S_1 were to be disjoint with respect to node N_2 in S_2, the same would be true of N_1 and N_2´s descendants in S, contrary to the assump-

tion that all nodes in S are pairwise conjoint.) Therefore, sooner or later, a reconciliation, R, between c_1 and c_2 will be called for.

Consider the scope, S´, of the auxiliary unifier or filter produced by R. S´ is formed by taking the union of S_1 with S_2 and deleting any nodes which are ancestors of others in the union. Because S_1 and S_2 each subsume S, so does S´.

If the reconciliation R fails, the resulting subsuming filter will make the original unification - the one seeking an evaluation of v - redundant and so the choice of candidate is of no consequence.

If the reconciliation R succeeds then any solution that includes the nodes in S necessarily includes those in S´. Although choosing c_1 rather than c_2 would, in general, give rise to different unifiers and hence different substitutions, both substitutions would include the unifier produced by the reconciliation R and this would guarantee that the application of either substitution to the expression ´v´ gives the same instance of that expression. Since v is an arbitrary variable, the same applies to all variables, and hence to an arbitrary expression.

In our example, the auxiliary unifier produced by R is {x/A, y/NIL}, whose scope is {N5, N7, N15}. If the evaluation of v were required in order to unify v with z.NIL and c_1 (whose term component is x.y) were chosen as the binding for v, the unifier produced would be {z/x, y/NIL}. If c_2 (whose term component is A.NIL) were chosen instead, the unifier would be {z/A}. The alternative substitutions take the form

1. {.. {.., v/x.y, ..}, {.., v/A.NIL, ..},
   {z/x, y/NIL}, {x/A, y/NIL}, ..}

2. {.. {.., v/x.y, ..}, {.., v/A.NIL, ..},
   {z/A}, {x/A, y/NIL}, ..}

respectively and it is readily verified that at least the extracts shown produce the same instance when applied to an arbitrary expression.

Now consider another situation, namely that no candidate binding for v exists at the time some unification (under scope S) seeks one. In this case, assuming the unification ends successfully, a unifier U of scope S will be produced and will include a binding, c: <t, S> for v, which will be registered in the normal way.

Suppose now that another binding for v, c′: <t′, S′> is made and that had it appeared earlier, c′ would have been a candidate binding for v in the unification producing U. One would like to be assured that the end result of the computation is not affected in any way through the late appearance of c′ and indeed this assurance is present:-

Since c′ would, by assumption, have been a candidate binding had it appeared earlier, we know that S′ subsumes S. Therefore, the nodes in S and S′ are pairwise conjoint and a reconciliation between c and c′ is required, the scope of the auxiliary unifier being S (because all nodes in S′ are ancestors of nodes in S).

Should this reconciliation fail, a filter of scope S will be established and this will preclude the appearance of U in any substitution.

On the other hand, if the reconciliation of c with c′ succeeds, any substitution which includes U will also include the auxiliary unifier resulting from the reconciliation because its scope is the same as U′s. Had the binding c′ arrived earlier, the components in the auxiliary unifier would have been included in the first unifier and no reconciliation would have been called for.

By way of illustration, suppose that the unifier U of scope S contains the component v/A.NIL and that this is because no candidate binding for v was found at the time unification took place. The binding c : <A.NIL, S> will appear in v′s register. Suppose further that at some later time, the component c′: <x.y, S′> is registered and that

this binding would have been a candidate binding for v in the unification that produced U, had it appeared earlier.

Because of this candidacy, we know that S´ subsumes S. Had this later binding appeared first, U would have included the components x/A, y/NIL. However, because of the late arrival of c´, U includes the binding v/x.y instead and a reconciliation between c and c´ is needed. This succeeds with the auxiliary unifier {x/A, y/NIL} (assuming that x and y are not bound) whose scope is also S. Hence any substitution that contains U also contains this auxiliary unifier and so the net result is the same as in the other case, where bindings for x and y appear as components of the original unifier.

6.6.2.2 Parallel Unification

Notice that the criterion for candidacy is satisfied if the binding recognised as a candidate was made earlier in the same unification – for in that case, both scopes will be the same.

The above arguments, showing the indifference of timing considerations when seeking a candidate, apply equally well in this special case. This makes it possible to concurrently match pairs of terms in a single unification.

For example, if f(A, u) and f(x, x) are to be unified (under some scope S), it is possible to unify the pairs of terms <A, x> and <u, x> concurrently.

If the binding <A, S> is (x-)registered before the other term unification seeks a binding for x, the unifiers will be {x/A} and {u/A}, both of scope S.

If the binding <u, S> is (x-)registered before the other term unification seeks a binding for x, the unifiers will be {x/u} and {u/A}, both also of scope S.

If neither of these timings pertain, that is, the (x-)register is empty when both term unifications seek a binding, then the unifiers {x/A} and {x/u} will result. Since both of these have scope S, a reconciliation between the two bindings for x will be called for and the auxiliary unifier {u/A}, also of scope S, will result.

In all three cases, any substitution whose corresponding solution includes the nodes of S, will have the same effect when applied to any expression involving x or u.

Notice that the earlier modification for producing Network behaviour is not compatible, as stated, with parallel unification.

6.6.3 Devolution of Processing

6.6.3.1 Ownership of Unifiers and Filters

We wish to avoid centralisation of all forms and this leads us to the concept of unifier and filter ownership. By this means we will be able to distribute unifiers and filters throughout the and-or tree and thereby localise certain operations on them.

Suppose S is the scope of a unifier or filter. Then the nodes of S are pairwise conjoint and it is easily verified that there is a unique head node which is their common ancestor but no descendant of this node has the same property.

We will call such a node the **owner** of the unifier or filter and will also refer to it as the nearest common ancestor of the nodes in S. We intend that unifiers and filters be accessible from their owning node.

Note that the term 'owner' has the expected connotations in the context of a primary unifier.

6.6.3.2 Filter Subsumption Check.

In later parts of this chapter, there will be calls to determine whether, given a scope S, there exists a filter whose scope S' subsumes S and we now show, in implementation terms, how such a check might be made.

Suppose we are presented with the scope $S = \{N_1, N_2, \ldots, N_n\}$ and we wish to know whether there exists a filter whose scope $S' = \{M_1, M_2, \ldots, M_m\}$ subsumes S.

If such a filter exists, its owner, M, will be the node which is the nearest common ancestor of M_1, M_2, ... and M_m.

Each of M_1, M_2, ..., M_m would be, under the supposition that S' subsumes S, an ancestor of at least one node in S. Define a set of nodes S" by

$$S'' = \{\text{nodes } N \text{ in } S \mid N \text{ descends from a node in } S'\}.$$

Then M is also the nearest common ancestor of the nodes in S".

Therefore M is a descendant of N, the nearest common ancestor of nodes in S.

Thus if such a filter exists, its owner will lie on that part of the and-or tree between the nodes of S and their nearest common ancestor, the node which owns the filter or unifier whose scope is S. Since all filters will, by assumption, be accessible through their owners, it will be possible to test whether the scope of any of these filters subsumes S (needless to say, these tests are most appropriately carried out concurrently with one another) and thereby decide

whether a subsuming filter exists. In this way, the search for a subsuming filter becomes a localised operation.

The above argument is illustrated by reference to Figure 24 on page 142 with S = {N5, N7, N14, N15} and S′ = {N14, N15}. The nearest common node N of S is N1 and a subsuming filter may be owned by any node along the branches between N1 and the nodes in S. In fact, node N9 would be the owner of a filter with scope S′.

We will also have to deal with the converse case, that is, given a filter with scope S′ owned by the node M, to identify those nodes of the and-or tree which might own a filter or unifier whose scope S is subsumed by S′.

The reasoning above indicates that such filters and unifiers would be owned by nodes lying along M′s ancestral branch and would consequently be sought along that part of the and-or tree.

6.6.3.3 Scope Subsumption

In this section and the next, we will be concerned with steps needed to test for scope subsumption and scope conjointness. We give Prolog algorithms based on particular representations of node names and scopes but do not mean to imply by doing so that a practical implementation will necessarily be restricted to them. These algorithms are conceptual and are presented for illustrative purposes.

We first need to fix a representation for node names.

Nodes are named by lists of integers. The root node - corresponding to the initial goal - is named by the empty list, NIL. If a node named by the list L has n child nodes, these are named by the lists 1.L, 2.L, ..., n.L (the order of naming is arbitrary).

We may also take advantage of the natural ordering of integers and use it to order nodes names, analogously to the ordering of branch

names in the Or-parallel proof procedure. Given two nodes N and N´, we consider their names to be reversed, that is we represent the names as [I_1, I_2, ...] and [I_1´, I_2´, ...] respectively (so that I_1 relates to the root node's child etc.). Integer pairs <I_1 I_1´>, <I_2, I_2´> are sequentially compared until a discrepancy, $I_k \neq I_k$´ arises.

Then $N < N$´ if $I_k < I_k$´ and N´ $< N$ if I_k´ $< I_k$. If neither condition applies, that is, one or both of the lists becomes exhausted, no ordering pertains.

Geometrically, if the and-or tree is depicted to reflect this ordering, then $N < N$´ iff N lies to the left of N´. For our purposes, we do not need to extend the ordering to cope with the case of one node descending from the other because in the context of a scope, where this ordering will be exploited, this possibility does not arise.

We consider a scope to be represented by an ordered list of node names (integer lists), and we will term it a scope list.

To show that scope S is subsumed by scope S´, it is necessary to show that each node in S´ is an ancestor of some node in S.

This ordering of nodes in scope lists allows the subsumption test to cycle through both scope lists in a co-ordinated manner, as indicated in the following program.

```
S is-subsumed-by NIL

N.S is-subsumed-by N´.S´ <- N < N´ &
                            S is-subsumed-by N´.S

N.S is-subsumed-by N´.S´ <- N´ is-an-ancestor-of N &
                            S is-subsumbed-by S´
```

where the second clause skips out unrelated nodes of the supposedly subsumed scope.

It is quite conceivable that a more advanced architecture will be capable of performing these tests more efficiently, possibly by operating on a natural - e.g. geometrical - representation of the and-or tree. For instance, the human intellect, when presented with the picture of such a tree and asked to determine whether N < N', would do so by glancing at the two-dimensional representation of the tree. An architecture that supported such a representation would be correspondingly effective.

6.6.4 Determination of Conjointness/Disjointness

With the above node naming scheme, it is readily verified that goal nodes have even length names and head nodes have odd length names.

Given two nodes named by the lists l1 and l2, the following algorithm determines whether the two nodes are conjoint or disjoint. (Here, the 'Reverse' relation holds if one list is the reverse of the other).

```
Node-relationship(l1, l2, r) <- Reverse(l1, l1') &
                                Reverse(l2, l2') &
                                Nr(l1', l2', r, DISJOINT)

Nr(NIL, l, r, r)
Nr(l, NIL, r, r)
Nr(n1.l1, n2.l2, r, r) <- ¬(n1 = n2)
Nr(n.l1, n.l2, r, flip) <- Switch(flip, flop) &
                           Nr(l1, l2, r, flop)

Switch(CONJOINT, DISJOINT)
Switch(DISJOINT, CONJOINT)
```

The algorithm considers common nodes, beginning at the root node, either until one of the two submitted nodes is encountered - in which case it is the ancestor of the other - or until a pair of distinct

ancestors is encountered. In the latter case, the two nodes are conjoint/disjoint depending on whether they have an even/odd number of common ancestors (including the root node) - i.e. depending on whether the last clause of the Nr program was used an odd/even number of times.

6.6.5 Registers

We mentioned earlier that a register exists for each variable introduced into the refutation. As one might expect, its use is an extension of that found in the Or-parallel proof procedure.

In the earlier scheme, the register was used to hold alternative bindings made in different parts of the search tree. A corresponding statement is true here, although, of course, the tree in this case is an and-or tree. There are two principal consequences of this, both resulting from the fact that the register might contain different bindings for the same variable.

Firstly, seeking an appropriate binding from the register is no longer deterministic; any candidate binding for the scope in question will do.

Secondly, and more importantly here, such alternative bindings need to be reconciled and this is most conveniently organised as part of the registration procedure, as we now describe.

6.6.5.1 Registration

The function of reconciling alternative bindings for the same variable is carried out by **reconcilers**. If a register already contains n bindings and a new one arrives, then n reconcilers are established,

each charged with the task of reconciling the new binding with one of the older ones.

6.6.5.2 Reconciliation

The reconciler is presented with two bindings, $\langle t_1, S_1 \rangle$ and $\langle t_2, S_2 \rangle$.

Its first task is to determine whether S_1 and S_2 are conjoint scopes.

If the scopes are not conjoint, the reconciler terminates.

If the scopes are conjoint, the reconciler will initiate an attempt to unify the terms t_1 and t_2 but first, for reasons which will be made clear in due course, it needs to compute the scope of the resulting auxiliary unifier (or filter) and its owner. Once it has done this, it establishes an empty auxiliary unifier and associates it - by some unspecified means - with the owner.

It then seeks a filter which subsumes the computed scope and if it finds one, it knows that the unification it is about to embark on is redundant. In this case, the reconciler terminates, first discarding the empty unifier.

Assuming that no such filter is found, the reconciler sets about unifying t_1 and t_2.

If the unification fails, a filter will be produced in place of the auxiliary unifier. The earlier text indicated that considerable computation may then need to be undertaken, specifically, promotion, reduction and pruning. Moreover, any unifier or filter whose scope is subsumed by the new filter may safely be discounted. All these aspects are discussed in the next section.

If the unification is successful, the bindings will be sent for registration as in the Or-parallel scheme. The processing described above will then be repeated for each new binding.

In fact, as we hinted at earlier, it is logically acceptable, although perhaps not feasible in practice, to register bindings as unification proceeds, rather than waiting for it to terminate as in the Or-parallel scheme. This relaxation is possible because not registering a binding in time will have the effect of inducing a new reconciliation - as described in the section on parallel unification.

After unification ends, the reconciler terminates.

6.6.6 Filter Incorporation

Incorporation of a filter, that is, taking steps to bring the information it carries into account, is essentially a two phase operation.

Firstly, an attempt should be made to simplify the filter in accordance with the earlier sections on promotion and reduction. If promotion is possible, a new filter will result - in which case an attempt should be made to simplify that filter in a similar way and so on until no further simplification is possible.

The result of this first phase is a filter, not necessarily the original one, which is input to the second phase. In the previous section on reconciliation, we pointed out that before a unification is begun, a check is made to determine whether a subsuming filter exists - and if it does, the reconciliation need not be done. Here, we need to consider the other timing eventuality, namely the production of a subsuming filter during or after such a unification. The second phase seeks to abandon active unifications and completed unifiers and filters if their scopes are subsumed by that of the new filter.

We now describe these two phases in the context of unifier and filter ownership.

6.6.6.1 Promote and Reduce Phase

We briefly recap the conditions under which promotion is possible.

Let F_n be a new filter and suppose that N is its owning node. Promotion is possible if other filters F_1, F_2, ..., F_{n-1} either exist explicitly or are implied by subsuming filters, providing that

1. the scopes of the n filters have all but one node in common - i.e. their intersection has order one less than that of each of the individual scopes - and

2. the n nodes outside this intersection are all the n children of the same parent node.

Promotion may then take place and the resulting filter is the above intersection augmented by the grandparent node. It is readily verified that if the filters F_1, F_2, ..., F_n exist <u>explicitly</u> they will all be owned by the same node (N).

Notice, however, that a filter produced by reduction may be owned by a proper descendant of the original filter's owner. An illustration, based on the and-or tree in Figure 39 on page 187, is the promotion of the filters { {B, F, G}, {C, F, G} } (both filters owned by A) to the filter {A, F, G} (still owned by A), followed by the latter filter's reduction to the filter {F, G} (owned by D).

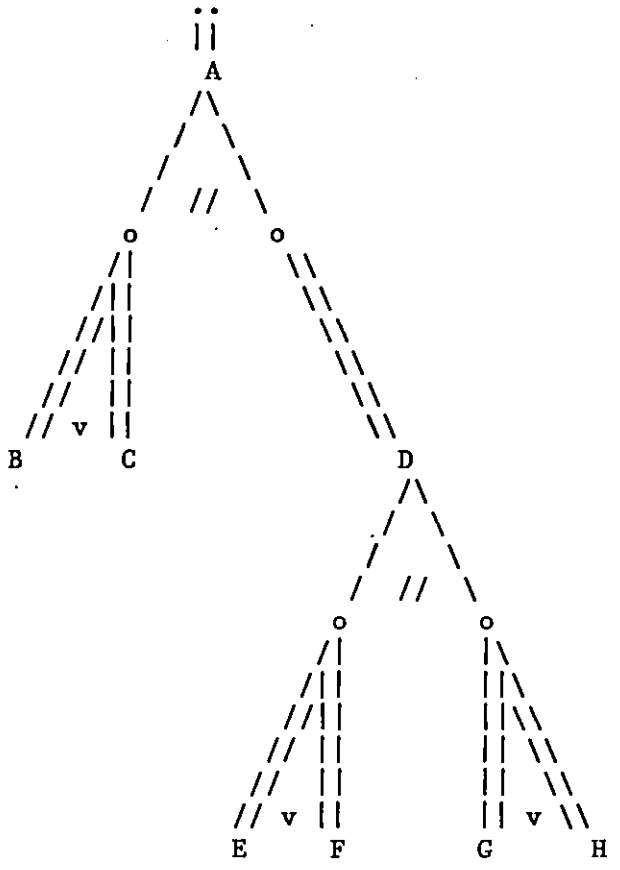


Figure 39.

Complications do arise, however, in considering promotions when one or more of the filters F_1, F_2, ... F_{n-1} is implicit - and conversely, when the filter F_n implies others (implication, in both instances being through scope subsumption). We will discuss these complications after the next section.

6.6.6.2 Filter Implementation Phase

Here our concern is to implement those simplifications to the overall computation made possible by the production of some new filter. The status of these operations should be borne in mind. They are not

necessary to preserve correctness (although neglecting to perform them may affect termination). They seek to save resources by abandoning unproductive paths but in doing so, they themselves may use significant resources. Analogous situations prevail in other areas of intelligent computation such as intelligent backtracking and the detection of clause subsumption.

The earlier section on the filter subsumption check indicated that any unifiers and filters whose scopes are subsumed by the new filter will be owned by nodes in a restricted part of the and-or tree and consequently, any searching for such unifiers and filters is restricted to that part of the tree.

In the special case of the new filter having a singleton scope, the implementation of that filter will also take on the responsibility for putting into effect the previously described branch pruning operations.

We comment here on the timing considerations concerned with this phase. Our anxiety here is the occurrence of two 'simultaneous' events:-

1. The search for a subsuming filter, carried out by a reconciler immediately before it makes its unification attempt and

2. The production of the subsuming filter which makes that unification attempt superfluous.

We might be concerned that the relative timing of these two events allows the subsumption to go by undetected.

In fact, this is not the case. The reason is that before its check, the reconciler establishes an empty unifier. Also, before its search for subsumed unifiers and filters, the implementation phase exhibits the filter with which it is concerned so that searching reconcilers may find it. It is evident therefore that if subsumption comes about then either the reconciliation finds the subsuming filter it seeks or the implementation phase finds the empty unifier it seeks

(or both). There is no possibility of an excluded middle that allows the subsumption to go by undetected.

6.6.6.3 Weaknesses in Filter Manipulation

The implementation proposal we gave for filter promotion is not complete, for it assumes that all filters input to it are explicitly held. In fact, not only is it incomplete, but as stated, it is not efficient. We will show why with the assistance of an example, which relates to Figure 39 on page 187. In the example, we assume the prior existence of filters {B, E, G} and {B, H} (both owned by A) and the production of the new filter {F, G} (owned by D).

Between them, these filters imply the filter {B} which in turn allows the branch leading to node B to be pruned. The derivation is illustrated in Figure 40 on page 190.

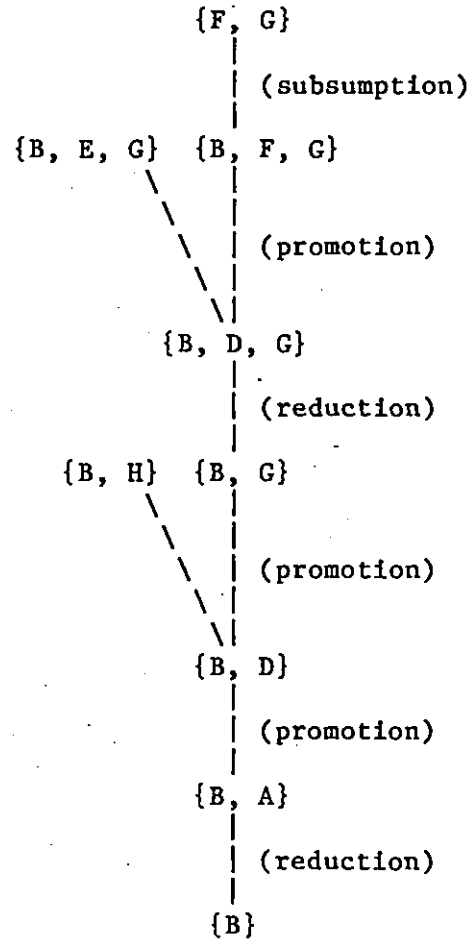


Figure 40.

It will be seen that the derivation is bottom-up. Worse, the filter promotion algorithm itself and the search for a subsuming filter are also bottom-up operations.

What one would like are top-down, goal-directed algorithms. This is very much an area of further research but we feel that an eventual solution will be along the following lines.

We suggest that (meta-level) goals should exist, one for each head node, each goal taking the form

<- Filter({node})

(conjecturing the non-existence of the singleton filter).

The meta-level clauses used in the sought derivations will be assertions of the form Filter(S) together with the following implications (augmented by suitable lower-level definitions).

Subsumption

```
Filter(s) <- Subsumes(s', s) &
             Filter(s')
```

(A filter of scope s exists if s' subsumes s and a filter of scope s' exists.)

Promotion

```
Filter(s) <- Split(s, n, s') &
             Child(n, n') &
             Children(n', nodelist) &
             Filterlist(s', nodelist)

Filterlist(s, NIL)
Filterlist(s, n.rest) <- Filter( sU{n} ) &
                         Filterlist(s, rest)
```

(A filter of scope s exists if s can be partitioned into a singleton {n} and residue s' and filters with scope s'U{n"} exist for each child, n", of n', where n' is some child of n.)

Reduction

```
Filter(s) <- Member(n, s) &
             Ancestor(n, n') &
             Filter( sU{n'} )
```

(A filter of scope s exists if there exists a filter whose scope comprises the nodes of s together with a (proper) ancestor of some node of s.)

We envisage that control over these derivations will be data driven. In other words, we imagine these derivations to remain dormant until an 'appropriate' filter is produced, this stimulating the continuation of the proof in corresponding parts of the search tree.

Thus for example (refer to Figure 39 on page 187), we envisage the meta-level goal <-Filter({B}) (associated with node B) to remain dormant until, for instance, Filter({B, H}) is produced. This would stimulate, by some appropriate reasoning, the development of the search tree to the stage indicated in Figure 41.

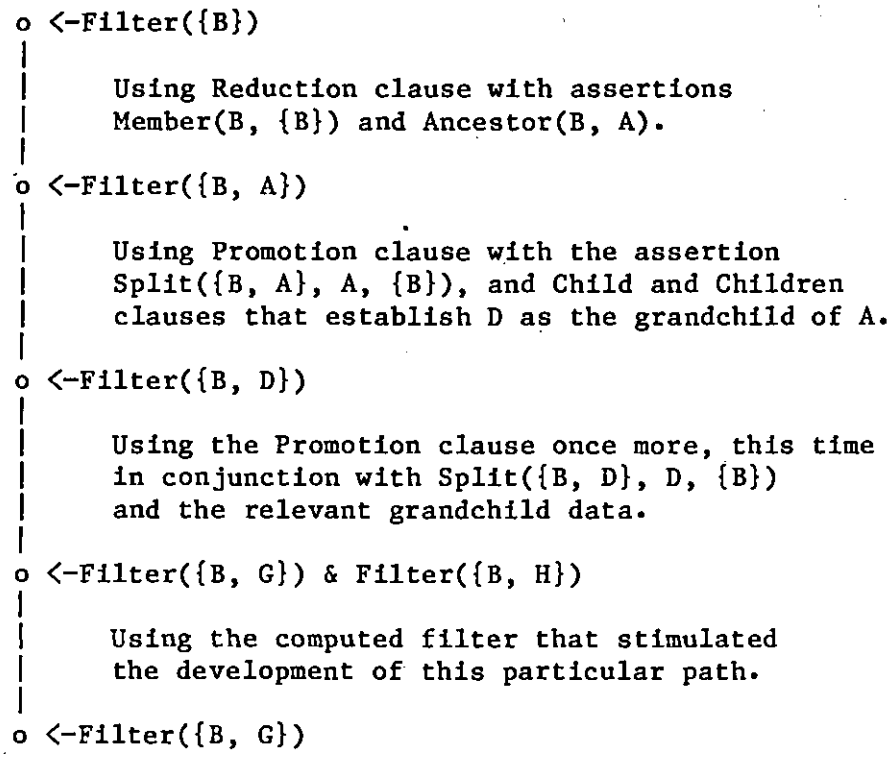


Figure 41.

The arrival of other filters would extend the search tree as appropriate.

6.7 ASSESSMENT

Because the research reported here is still in its early stages, this assessment will necessarily be inconclusive. Nevertheless, we feel that such an exercise is useful because it summarises the main features of our proposal whilst indicating areas of further research.

6.7.1 Degree of Parallelism

We believe that the scheme we propose has as much potential for parallelism as is possible for Horn clause programs, for it allows for both of the high-level forms of parallelism together with parallel unification. Of course, in practice, some restraint will generally be necessary and we have indicated ways of achieving this end. Ideally, an automated means of control is needed and we see the provision of such a mechanism as one aspect of future research. Adding control advice by means of input-output mode declarations may well be a suitable intermediate solution. In the short term, restriction of and-parallelism to those parts of the overall computation known to give rise to independent subgoals seems a worthwhile first step.

6.7.2 Termination

One weakness of our scheme, inherited from the naive scheme albeit in a modified form, is that solutions of the user's goal cannot be extracted until the computation as a whole terminates. This is a consequence of the notion of candidacy, which we may re-formulate here as stating that a candidate solution is not a solution until it is known that no filter precludes it. Unfortunately, the scheme we have described does not tell us when all filters which might potentially subsume any particular solution have been computed. Instead, we must

rely on the cessation of all activity (as in the naive scheme) and then produce solutions by generating and filtering candidate solutions.

The weaknesses of this approach are quite severe, for a computation which gives rise to one or more infinite branches of the and-or tree (after pruning is taken into account) will fail to return any solutions. A crude way of overcoming this deficiency is to pre-set a limit on the depth of nodes in the and-or tree and to treat nodes beyond that depth as nodes of failure. This may be acceptable in certain cases, possibly database applications, although in general, we consider this method as no more than a temporary expedient since it disturbs the semantics of the programming language.

We do not see the provision of a scheme which extracts solutions as they are computed as an insurmountable problem. What is required is some mechanism to co-routine the extraction of solutions with the primary and auxiliary unifications concerned in their generation. This indicates the need for some form of feedback from reconciliations so that when particular reconciliations succeed, the success is made known to the extraction mechanism.

We are unable to be more specific at this stage except to add that comparisons between our proposal and other more familiar schemes operating under certain execution controls might supply the necessary clues. This would in any case be a useful exercise in its own right and we would be particularly interested in establishing the relationship between our scheme and the Connection Graph model [26].

6.7.3 Control

We have already indicated the need for further general research into the control of program execution. Here, we will just say that we consider the weakness in control is caused by the same underlying deficiency as that mentioned in the section above viz. the absence of sufficient feedback. To be more specific, we are referring to the

difficulty which arises when a clause whose body takes the form S_1 // S_2 // ... // S_n is invoked in response to a goal and the residual sequence S (the goals outstanding at the node less the selected one) is non-empty.

We require some means of co-ordinating and feeding the 'results' of the computation S_1 // S_2 // ... // S_n to the sequence S so that goals in S are not selected too early.

6.7.4 Filter Manipulation

For the sake of completeness here, we mention once more the outstanding problem of providing a top-down method of incorporating filters.

6.7.5 Architecture

The degree of concurrency allowed in our scheme, arguably the maximal possible for Horn clause programs, would typically require more computational power than is available today: the Japanese research effort into fifth-generation computers [45] might reasonably be expected to provide suitable hardware. The timescales for carrying out the outstanding research should therefore be viewed in this context.

We have not addressed the area of architecture in the proposal although it is worth emphasising here the need to provide a machine sufficiently powerful to overcome the extra workload of the And-or scheme. As an indication of this overhead, we might consider the task of looking up a binding. In the Or-parallel scheme, this reduces in essence to the straight-forward comparison of two bit strings; in the And-or scheme, a test for scope subsumption is the corresponding operation. Clearly an architecture suitable for the And-or scheme would

need to be considerably more advanced than the one we described for its Or-parallel counterpart.

CHAPTER 7: CONCLUSION

7.1 RELATED RESEARCH

Our research seeks to apply parallel computation to Horn clause programs and as such attempts to bring together two major areas of investigation, viz. Parallel Computation and Logic Programming.

Much research effort is currently being applied into ways of exploiting the potential for parallelism that new technology makes possible and references were made to such researches in earlier chapters. The closest field of research to ours is that concerned with the parallel execution of functional programs - after all, the formalism of Horn clauses and those of functional programming languages share many characteristics, not the least of which being the parallel execution potential of programs expressed in those formalisms. However, the relationship is not as close as one might wish, for the mechanics of actually achieving parallelism - and that is the primary research topic of this thesis - are dissimilar.

Execution of a functional program may be viewed as the reduction of a corresponding expression graph [13] whereby concurrency is achieved through the parallel reduction of expressions. Although this concurrency is the equivalent of and-parallelism, expression graphs in themselves have no direct relevance to Logic program execution which is based, as previously described, on resolution.

Where the two fields of research are likely to be most closely related is on the architecture side. Our proposed architecture for the Or-parallel scheme is, at a superficial level, very close to that put forward by Darlington and Reeve [13]. Both proposals, for example, distribute "packets" by means of a ring and both have access to shared memory. Where the two schemes differ is in the nature of the processing elements themselves - for our proposal distributes the global memory throughout the PE's whereas theirs has shared access to a common packet pool through a packet pool controller. It is not

inconceivable that a common architecture might unify the two proposals.

Whilst on the topic of architecture, we should mention here the work of Rieger, Bane and Trigg [36]. ZMOB is not specialised to logic but is intended as a general multi-processor for use in a variety of applications, as described in [37]. Our belief, as discussed earlier, is that ZMOB will not support the level of storage access that our scheme requires and it will be interesting to note how the difficulties we foresee are overcome. In this context, we believe that Minker is planning an implementation of Prolog on ZMOB.

Conery and Kibler's AND/OR proposal [11] is in some respects similar to our Or-parallel scheme. For the main part here, we will only concern ourselves with those parts of their scheme above the level of the architecture.

Their model, as its name suggests, is based on the and-or tree view of computation. They define two sorts of processes, AND and OR processes, which they associate with AND and OR nodes of the tree respectively, one process per node. Thus AND processes have OR processes for their children and vice versa. We first describe the operation of an AND process.

An AND process is presented with a conjunction of goals to solve and initially establishes an OR process with the task of finding solutions of the first of these. It then suspends itself awaiting the arrival of one such solution. (Thus there is no and-parallelism - although Conery and Kibler refer to future plans for investigating the incorporation of such parallelism whenever it is known that subgoals share no variables.)

Assuming that a solution ('substitution') is returned by the child OR process, the AND process becomes active once more by applying the substitution to the remaining subgoals and creating an OR process to solve the second goal, and so on. When no further subgoals remain, the substitution constructed by composition of the solutions returned

by the child OR processes is sent to the parent OR process as a solution of the conjunction.

If, however, a child OR process fails or indeed the parent of the AND process asks for an alternative solution, the AND process requests the preceding OR child process to compute another solution. If this is not possible because the OR child which failed was charged with solving the first goal in the conjunction then the AND process itself fails.

We now describe the operation of an OR process. An OR process is supplied with an atomic goal and has the task of finding a solution which is acceptable to its parent AND process. It begins by attempting to locate a clause whose head matches the supplied goal. If such a clause is found and turns out to be an assertion, the solution returned to the parent is simply the matching substitution. On the other hand, if the clause is an implication, the matching substitution is applied to the body of the implication and an AND process is established to solve the conjunction of subgoals thus formed.

The OR process continues by attempting to unify the goal with the heads of all remaining (applicable) clauses, for it may well happen that the parent AND process will find the first solution returned unacceptable and request another. In this way, the AND/OR scheme seeks to exploit all the or-parallelism implicit in the user's program.

On receipt of a solution found by a child AND process, the OR process composes it with the matching substitution saved from the earlier unification and thus forms a solution of the goal it was asked to solve. Once the first such solution has been sent to the parent AND process, the OR process locally saves any further solutions and passes them one by one to its parent - but only when it is specifically requested to do so. The scheme incorporates a degree of optimisation insofar as OR processes suppress solutions which are identical to those previously sent to the parent. (Other areas of optimisation are also described.)

Having outlined their scheme, we summarise below the principal similarities and differences between their approach and ours.

Similarities:-

Both schemes exploit or-parallelism only.

Both schemes are based on a multi-processor architectures. Each processor holds a local copy of the user's program.

Differences:-

Their design is based on explicit application of substitutions to outstanding goals. Ours is based on structure-sharing of goal lists and of bindings (via registers).

Their PE's only communicate by means of messages. Substitutions and fully instantiated lists of goals are examples of messages communicated from PE to PE.

Each of their processes is entirely executed on the PE that first accepts the process. In our scheme, PE's are considered to be equal computational resources, each capable of continuing any process at any point.

Robinson and Sibert's work on Loglisp [40] is of interest here because its organisation is not based on the LRDF search strategy. Loglisp is a marriage of Horn clause logic and Lisp which seeks to exploit the control facilities of the latter whilst retaining the "pure" characteristics of the former. In this way, extra-logical features, as found in most implementations of Prolog (and as criticised in [32]; replied to in [14]), may be separated from the logic itself and implemented instead through the Lisp component.

Although not primarily put forward as a scheme for exploiting parallelism, the Loglisp proposal seems suitable for such application: for the decision to abandon LRDF search means that some method of pursuing alternative derivations - at least in quasi-parallel - has to be devised. The chosen approach, that of representing alternative bind-

ing environments by means of association lists, differs from our Or-parallel proposal of having registers of bindings (their scheme, like ours, structure-shares bindings). We will describe our understanding of their mechanism to enable a comparison of the two alternatives to be made.

The starting point in our description is a conventional environment of activation records. It will be seen that the modification accommodates separate derivations by arranging for environments to be copied whenever necessary (the copying is not naive). Because derivations are separated in this way, it is possible to store bindings uniquely in the activation record where the variable is introduced - although a minimal amount of searching is still required and so access is not as direct as in a conventional implementation.

In this exposition, we will simplify matters by considering an activation record to consist of nothing other than the contained bindings i.e. we will drop all references to the other items of these records. Figure 42 on page 202 illustrates the first modification they make.

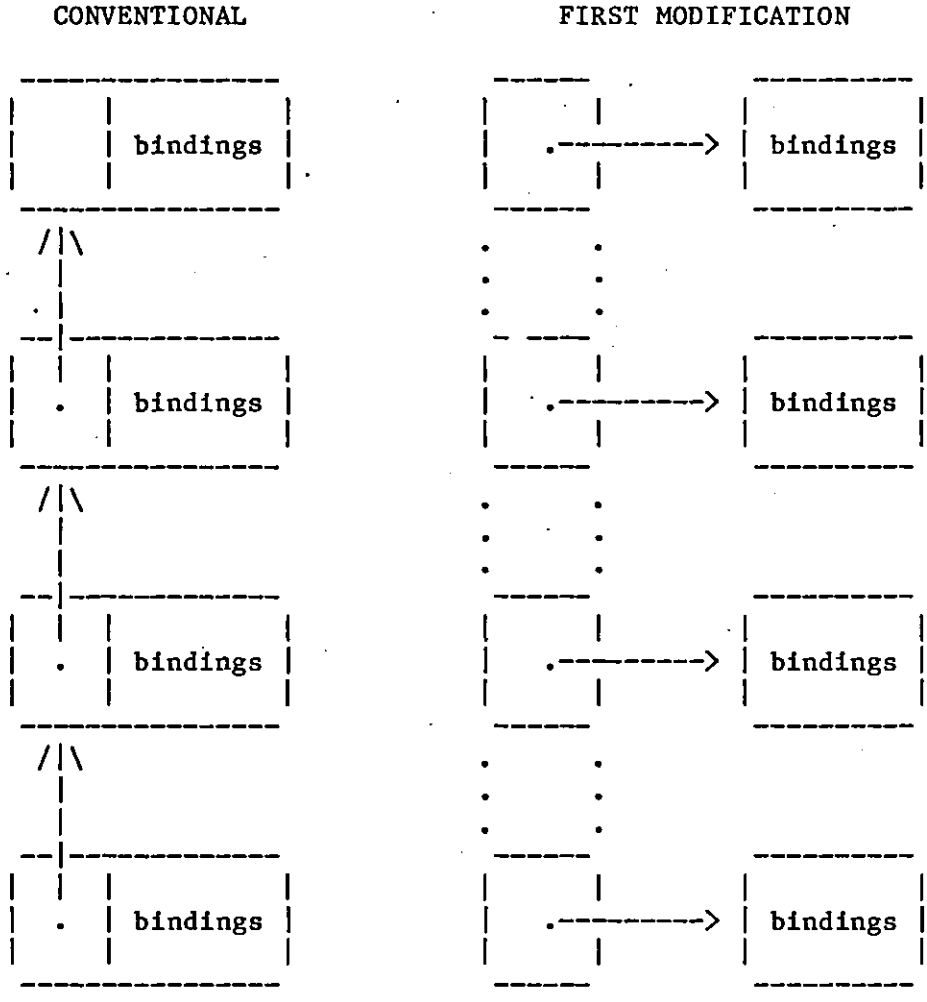


Figure 42.

The modification splits the bindings from the rest of each activation record by means of a "spine" of pointers. In conventional implementations, the varying amount of storage reserved for bindings (amongst other things) dictates the need to construct stacks by means of pointers connecting neighbouring activation records. In Robinson and Sibert's scheme, an environment is represented by a spine and since each vertebra is of the same size, the spine is implemented as an array.

The second modification implements the bindings in each activation as a chain, the association list, rather than by the (pre-reserved) array of binding space found in conventional implementations. Prolog's classic direct access to a binding is thus lost; instead a search through the appropriate association list now becomes necessary. If a variable is unbound in a particular environment, no entry exists for it in the relevant association list. An activation record and association list are shown in Figure 43.

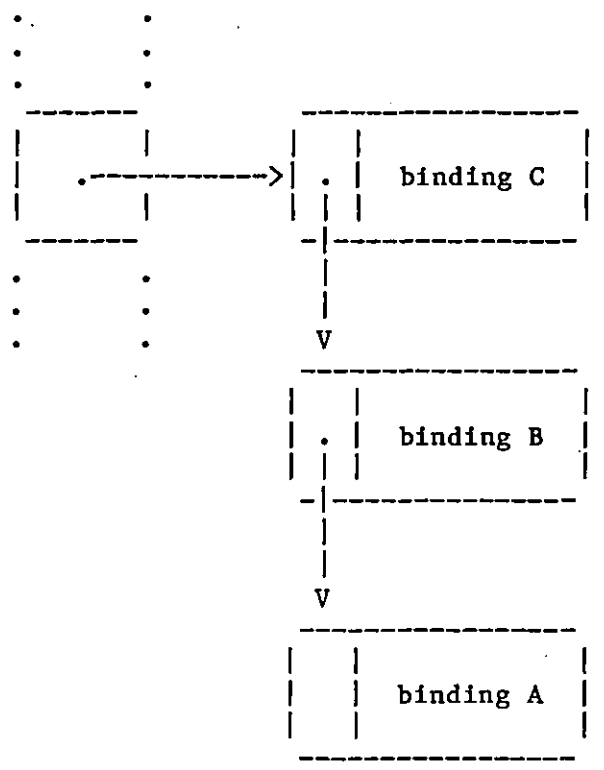


Figure 43.

Having indicated the change made to a single environment, we are now in a position to describe how structure-sharing of bindings is brought about.

To copy an environment, one copies the array which implements the spine. Figure 44 on page 204 illustrates sections of two environments that share three bindings.

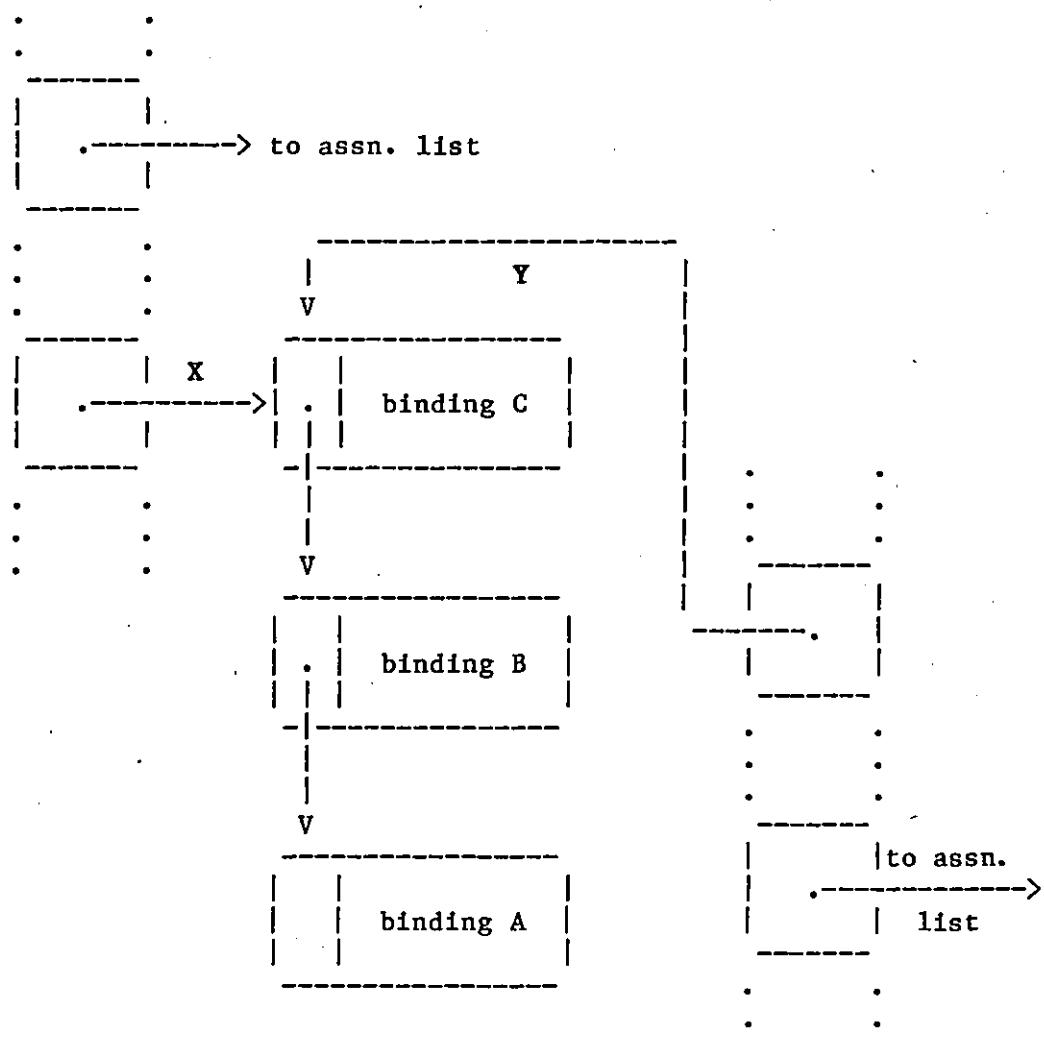


Figure 44.

This copying of spines is typically done whenever the search tree forks. Let us consider just one derivation, the one whose environment is represented by the left spine in Figure 44. Suppose that a binding D is made in that derivation and the variable thus bound was introduced in the same activation as the variables appearing in the bindings A, B and C. The new binding is then chained in at the point X. Likewise, a new binding (possibly for the 'same' variable) made in the environment represented by the right spine is chained at Y. It should be evident that bindings A, B and C are thus shared between both environments.

This approach to holding bindings involves copying the current spine at each forking node in the search tree. Since the number of vertebrae is equal to the depth of the forking node in the tree, the overhead of such copying is not bounded by any constant. Our scheme does not have a similar overhead.

When it comes to accessing a binding, some searching is involved in both schemes. In our's, the variable's name directs the search to the appropriate register, which is then examined. Robinson and Sibert instead seek a binding in the appropriate association list. That search is bounded by the number of variables introduced by the clause used in the relevant activation. In our scheme, the search is unbounded because the number of alternative bindings for the 'same' variable is not bounded. In practice, of course, one would not expect more than a few, usually zero, one or two, bindings - although, of course, this is entirely dependent on the application. We feel that a comparison of the two schemes by means of simulation would be a particularly interesting and instructive exercise.

A somewhat more established proposal, the $\overline{\Pi}$-representation formalism of Fishman and Minker [17], was mentioned in an earlier chapter and we include a reference to it here for the sake of completeness. Unlike the other proposals discussed in this section, it does not seek to exploit parallelism through parallel computation but rather through the choice of program representation.

The researches of Kibler and Conery, Robinson and Sibert and Fishman and Minker are concerned with or-parallelism; the research of Clark and Gregory [5], to which we now turn, seeks instead to exploit and-parallelism.

Their proposal complements those we have put forward because we have no And-parallel scheme as such. Clark and Gregory view the output of a computation as being in a relation with the input (rather than a function of it) and their scheme computes no more than one

instance of that relation. The particular instance found typically depends on the temporal behaviour of the various components that together comprise the computation.

Their proof procedure is based on don't care non-determinism, as found in Dijkstra's guarded commands. A clause in their language takes the form

$$P \leftarrow G \mid S$$

where G is the guard and S takes the form S_1 // S_2 // ... // S_n where each S_i is a sequence (G and/or S may be null). The guard G and each sequence S_i consists of a conjunction of atoms.

Declaratively, the clause is read as though the body were simply comprised of the conjunction of all atoms in G and each S_i. Operationally, the clause is a candidate clause for solving goals which match the head P, provided that the guard is a true conjunction of ground atoms. Moreover, once a candidate clause is discovered, its acceptance precludes the selection of any other candidate clause and computation is irrevocably committed to that choice. In this way, backtracking and other ways of coping with non-determinism are dispensed with. In turn this leaves the way open to an elegant design which implements each sequence as a process and establishes communication between such processes through channels, each channel corresponding to a shared variable (i.e. a variable appearing in more than one sequence). These channels have a definite direction of flow (under the programmer's control) and give rise to computations with a pipeline or network flavour.

It will be appreciated that the language differs from the language of Horn clauses both in form and semantics and so their scheme cannot be directly compared with either of our schemes. Their approach makes the formalism particularly suitable for applications which are functional apart possibly from a degree of don't care non-determinism whereas our proposals are quite general and in particular might be suitable for database applications.

We know of no research related to our And-or scheme.

7.2 FUTURE RESEARCH

7.2.1 Or-parallel Proof Procedure

At various points in our description of the Or-parallel scheme, we were obliged to make statements of the form "...in the absence of simulation..." and rely instead on intuitive analysis. Of course, analysis of a system is a valid method of gaining insight into its behaviour, provided that the model used accurately represents it. However, accurate representation of a complex computer system is a notoriously difficult task and for critical systems is usually supplemented by simulation in one form or another.

We feel such simulation is important in our scheme and in particular, we would like to investigate the relationship between the number of PE's and overall speed of computation. We believe that this relationship depends quite critically on the nature of the global storage interconnection network and would like to simulate various interconnection strategies. In particular, we would like to be assured that given a sufficiently powerful network (and a sufficiently "parallel" program), addition of extra PE's never increases the overall time of computation and, for large numbers (n) of PE's, decreases it in a manner close to the ideal (proportional to 1/n). We would also like to simulate the shared bus interconnection network and would be very interested in confirming our estimate of the maximum number of PE's that might be supported by this extremely simple and cheap (global) resource.

Another area of further research worth pursuing is that of optimisation. Our proposal has been kept as simple as possible but certain economies, for instance in the utilisation of store, may be possible. Such optimisations generally have a processing penalty and one would have to carefully evaluate whether they are worthwhile in the context of ever-decreasing memory costs, given that the primary motive for providing parallel execution in the first place is the desire to increase the speed of computation.

The final area of research is that of building a prototype system and evaluating it to confirm the analysis and simulation described above.

7.2.2 And-or Proof Procedure

The previous chapter contained an assessment that outlined areas of further research on the And-or proof procedure. In this section, we point out that in addition to those investigations, the further research described above for the Or-parallel proof procedure also applies to the more novel scheme, perhaps more so. We would also like to investigate the computational complexity of certain key aspects.

Finally, we feel that a certain degree of more theoretical research is called for. Our description has been based largely on intuition and we would like to see a more soundly based exposition of the scheme. In particular, we would like to see proofs of its correctness and completeness.

8.0 REFERENCES

[1] Bowen, K.A., Kowalski, R.A., [1981], Amalgamating Language and Metalanguage in Logic Programs. To appear in Logic Programming Papers (K.L. Clark and S-A Tarnlund eds.) Academic Press, New York.

[2] Boyer, R.S., Moore, J.S., [1972], The Sharing of Structure in Theorem Proving Programs. Machine Intelligence Vol. 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, eds.), pp. 101-116.

[3] Burstall, R. M., Darlington, J., [1977], A Transformation System for Developing Recursive Programs, J. ACM, Vol. 24 No. 1 (January 1977).

[4] Clark, K.L., [1978], Negation as Failure. Logic and Data Bases, (H. Gallaire and J.Minker eds.), Plenum Press, New York, pp. 293-322.

[5] Clark, K.L., Gregory, S., [1981], A Relational Language for Parallel Programming. Proceedings of 1981 Conference on Functional Languages and Computer Architecture.

[6] Clark, K.L., McCabe, F.G., [1979], The Control Facilities of IC-PROLOG, Research Report, Department of Computing and Control, Imperial College, London University.

[7] Clark, K.L., McCabe, F.G., [1980], IC-PROLOG: Aspects of its Implementation. Research Report, Department of Computing and Control, Imperial College, London University.

[8] Clark, K.L., Tarnlund, S-A., [1977], A First Order Theory of Data and Programs. Proc. IFIP 77, North Holland Publishing Co., Amsterdam. pp. 939-944.

[9] Codd, E.F., [1970], A Relational Model for Large Shared Data Bases. C. ACM, Vol. 13, No. 6 (June 1970), pp. 377-387.

[10] Colmerauer, A., Kanoui, H., Pasero, R., Rousell, P., [1973], Un Systeme de Communication Homme-Machine en Francais. Rapport, Groupe Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

[11] Conery, J.S., Kibler, D.F., [1981], Parallel Interpretation of Logic Programs. Proceedings of 1981 Conference on Functional Languages and Computer Architecture.

[12] Daglass, E.L., [1977], A Multiprocessor - CYBA-M. Information Processing 77 (B.Gilchrist ed.), IFIP North Holland Publishing Co., pp. 843-848.

[13] Darlington, J., Reeve, M., [1981], ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceedings of 1981 Conference on Functional Languages and Computer Architecture.

[14] van Emden, M.H., [1980], McDermott on PROLOG: A Rejoinder. SIGART Newsletter 73, Oct 1980, pp. 19-20.

[15] van Emden, M.H., Kowalski, R.A., [1976], The Semantics of Predicate Logic as a Programming Language. J. ACM, Vol. 23, No. 4, pp. 733-742.

[16] Farrell, E.P., Ghani, N., Treleaven, P.C., [1979], A Concurrent Computer Architecture and a Ring Based Implementation. Proc. Sixth International Symposium on Computer Architecture, 1979, pp. 1-11.

[17] Fishman, D.H., Minker, J., [1975], $\overline{\Pi}$-Representation. A Clause Representation for Parallel Search. Artificial Intelligence, Vol. 6, No. 2, pp. 103-127.

[18] Futo, I., Darvas, F., Szeredi, P., [1978], The Application of PROLOG to the Development of QA and DBM Systems. Logic and Data

Bases, (H. Gallaire and J.Minker eds.), Plenum Press, New York, pp. 347-375.

[19] Gallaire, H., Minker, J., (Editors) [1978], Logic and Data Bases. Plenum Press, New York.

[20] Hill, R., [1974], LUSH Resolution and its Completeness. DCL Memo No. 78, University of Edinburgh, School of Artificial Intelligence.

[21] Hoare, C.A.R., [1961], Algorithm 64. C.ACM, Vol.4, pp. 321.

[22] Hoare, C.A.R., [1978], Communicating Sequential Processes. C. ACM, Vol. 21, No. 8 (August 1978), pp. 666-677

[23] Hogger, C.J., [1979], Derivation of Logic Programs. Ph. D. Thesis, Imperial College, London University.

[24] Horn, A., [1951], On Sentences which are True of Direct Unions of Algebras. Journal of Symbolic Logic, 16, pp. 14-21.

[25] Kowalski, R.A., [1974], Predicate Logic as Programming Language. Proc. IFIP 74, North Holland Publishing Co., Amsterdam. pp. 569-574.

[26] Kowalski, R.A., [1975], A Proof Procedure using Connection Graphs. J. ACM Vol. 22, No. 4, pp. 572-595.

[27] Kowalski, R.A., [1979], Algorithm = Logic + Control. C. ACM, Vol. 22, No. 7 (July 1979), pp. 424-436.

[28] Kowalski, R.A., [1979], Logic for Problem Solving. North Holland, New York.

[29] Kowalski, R.A., Kuehner, D., [1971], Linear Resolution with Selection Function. Artificial Intelligence, Vol. 2, pp. 227-260.

[30] Loveland, D.W., [1970], A Linear Format for Resolution. Symposium on Automatic Demonstration, Lecture Notes in Math 125, Springer-Verlag, Berlin, pp. 147-162.

[31] Maller, V.A.J., [1979], The Content Addressable Filestore - CAFS. ICL Technical Journal, 1979 1(3), pp. 265-279.

[32] McDermott, D., [1980], The PROLOG Phenomenon. SIGART Newsletter 72, July 1980, pp. 16-20.

[33] Nilsson, N.J., [1971], Problem-solving Methods in Artificial Intelligence. McGraw Hill, New York.

[34] Patel, J. H., [1979], Processor-Memory Interconnections for Multiprocessors. Proc. Sixth International Symposium on Computer Architecture, 1979, pp. 168-177.

[35] Pease, M.C., [1977], The Indirect Binary n-cube Microprocessor Array. IEEE Transactions on Computers, vol C-26 (May 1977), pp. 458-473.

[36] Rieger, C., Bane, J., Trigg, R., [1980], ZMOB: A Highly Parallel Multiprocessor. Department of Computer Science, University of Maryland, College Park, MD 20742.

[37] Rieger, C., Trigg, R., Bane, B., [1981], ZMOB: A New Computing Engine for AI. Department of Computer Science, University of Maryland, College Park, MD 20742.

[38] Roberts, G., [1977], An Implementation of PROLOG. M.Sc. Thesis, University of Waterloo.

[39] Robinson, J.A., [1965], A Machine Oriented Logic Based on the Resolution Principle. J. ACM, Vol. 12, No. 1 (January 1965), pp. 23-41.

[40] Robinson, J.A., Sibert, E.E. [1980], Logic Programming in Lisp. School of Computer and Information Science, Syracuse University, New York.

[41] Roussel, P., [1975], PROLOG: Manuel de Reference et d'Utilisation. Group d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

[42] Treleaven, P.C., [1980], Workshop Report - VLSI: Machine Architecture and Very High Level Languages. SIGARCH Computer Architecture News, Vol. 8 No. 7 (Dec 1980), pp. 27-38.

[43] Warren, D.H.D., [1977], Implementing Prolog. Res. Rep. 39, 40. Dept. of A.I., University of Edinburgh.

[44] Weyrauch, R., [1980], Prolegomena to a Theory of Mechanized Formal Reasoning. Artificial Intelligence, 13, pp. 133-170.

[45] [1980], Interim Report on Study and Research of Fifth-Generation Computers. Japan Information Processing Development Center.

[46] [1980], Reference Manual for the Ada Programming Language. United States Department of Defense.

9.0 APPENDIX: DETAILS OF BASIC INTERPRETER

Demonstrate

The program for Demonstrate is a recursive procedure set whose base case is given by the failure to select a goal:-

```
Demonstrate(program, state)
  <- ¬Select(goal, state)
```

and whose recursive case is given by the clause

```
Demonstrate(program, state)
  <- Select(goal, state) &
     New-ar(program, state, goal, ar) &
     Demo(program, ar.state)
```

where the first call selects a goal from the current state and the second creates a "blank" activation record, ar. The resulting (intermediate) state is passed on to Demo.

The three-place Demonstrate referred to in Chapter 3 would instantiate the variable representing the final state to the input state in the base case (and pass it through to the Demo call otherwise).

The specifications of ´Select´ and ´New-ar´ are deferred for the moment.

Demo

There are two ways to solve Demo, depending on whether matching is successful or not.

```
Demo(program, state-in)
  <- Match(program, state-in, state-out) &
     Demonstrate(program, state-out)

Demo(program, state-in)
  <- ¬Match(program, state-in, state-any) &
      Demo´(program, state-in)
```

If matching is successful, the Demonstrate step succeeds and recurses to the next stage; if it fails, a call is made to Demo´ whose outcome depends on whether or not any further clauses are available to solve the activation record´s goal.

Note that in a conventional (destructive assignment) implementation the effects of a partially complete match would have to be undone so that the state could be returned to its pre-match condition. For the sake of simplicity (in the ´Match´ procedure set), we do not simulate this aspect but instead take advantage of the assignment-free nature of Prolog and reference the previous state, state-in, in Demo´.

Demo´

After a matching failure, subsequent computation depends on whether or not there are further clauses which might be invoked in response to the selected goal:-

```
Demo´(program, state-in)
  <- Next-clause(program, state-in, state-out) &
     Demo(program, state-out)

Demo´(program, ar.state-in)
  <- ¬Next-clause(program, ar.state-in, state-any) &
      Backtrack(program, state-in, state-out)
```

If no further clause can be tried, the top activation record is simply destroyed and ´Backtrack´ is called to operate on the remainder of the input state.

Backtrack

```
Backtrack(program, state-in, state-out)
  <- Un-match(state-in, state-out) &
     Demo'(program, state-out)
```

Backtracking operates by undoing the previously successful unification and then calling Demo' to try again with the next clause indicated in the top activation record. If no such clause exists, Demo' will call Backtrack again. Notice that if Backtrack fails, so too does Demonstrate' - i.e. in effect, the underlying interpreter is being used to implement failure to demonstrate.

The specification of 'Un-match' is deferred until after matching has been described.

We now specify the lower levels of the program.

Select

'Select' implements the selection strategy in the context of the shared goal list structure described in Chapter 3.

```
Select(goal, ar.state)
  <- First-subgoal(ar, goal)

Select(goal, ar.state)
  <- Assertive(ar) &
     Isolate(ar, goal*) &
     Select'(goal, goal*, state)
```

First-subgoal only succeeds if the clause referred to in the third component of the activation record 'ar' is an implication, in which case 'goal' is instantiated to a term which represents the goal derived from the first atom of the body.

The second definition applies only if the clause referred to in the latest activation record is an assertion, in which case, ´Isolate´ abstracts the second component of ´ar´, the term which represents the goal just unified.

The definitions of ´First-subgoal´, ´Assertive´ and ´Isolate´ are straightforward and detailed specifications are not given.

Select´

Select´ is presented with a state and goal (in goal*) and is to instantiate the term ´goal´ to the goal whose selection follows that of goal* in the left-right, last-in-first-out strategy.

```
Select´(goal, goal*, state)
   <- Find-state(goal*, state, state*) &
      Select*(goal, goal*, state*)
```

´Find-state´ (not given) derives the state whose top ar is that in which goal* is introduced - i.e. it "strips off" intervening activation records.

Select*

```
Select*(goal, goal*, ar.state)
   <- Next-goal(goal, goal*, ar)

Select*(goal, goal*, ar.state)
   <- ¬Next-goal(goal, goal*, ar) &
       Isolate(ar, goal´) &
       Select´(goal, goal´, state)
```

Next-goal succeeds if goal follows goal* in the calls introduced by the antecedent referenced in ´ar´.

The second clause applies in the event of goal* being the last goal derived from the antecedent of the clause referred to in 'ar', in which case the goal (goal') named in 'ar' is input to a recursive call of Select'.

New-ar

New-ar supplies a new activation record for the given goal:-

```
New-ar(program, state, goal, ar(level, goal, 1, UNKNOWN, NIL))
  <- Unique(state, level) &
     Possible(program, goal)
```

In a von Neumann implementation, the name represented by 'level' would be implicit and take the form of an address - that of the activation record. In this program, 'Unique' has the task of generating a level different from all those used in ar's embodied in 'state'. Levels are represented by natural numbers and a unique level is generated by incrementing the level of the top-most ar in 'state' by one.

The constant '1' indicates that the clause to be tried is the first in the procedure set for the supplied goal. Its analogue in a realistic implementation would be a reference pointer to the first clause in the procedure set, the establishment of this pointer in effect matching the goal and head predicate symbols. We simulate this matching of predicate symbols via the 'Possible' call which merely confirms the existence of the procedure set. The definition of 'Possible' is straightforward, given that programs are normally organised as collections of procedure sets.

The constant UNKNOWN, occupying the position of the bindings argument of the activation record, will later be replaced by a list, each item of which corresponds to a new variable and initially signifies that the variable is unbound. It is not known at this stage how many such entries are to be made since the clause has not yet been accessed.

Next-clause

The third item in the most recent activation records indicates the current procedure definition being tried. Initially it is 1. A call to Next-clause serves to produce a new state whose only difference from that supplied is that the third argument of the most recent activation record is incremented by one - but only if 'program' indicates that the corresponding clause exists. Note that this implicitly implements the matching of predicate symbols. This is because the predicate symbols of the goal and first procedure definition used were matched in 'New-ar' and 'Next-clause' merely perpetuates this matching by supplying clauses from the same procedure set.

The definitions are too detailed to give here.

Match

```
Match(program, state-in, state-out)
   <- Get-goal-terms(state-in, terms-goal) &
      Get-head-terms(state-in, program, terms-head, state-inter) &
      Match'(terms-head, terms-goal, state-inter, state-out)
```

As explained above, the goal and head predicate symbols will already have been matched but before matching (specified in Match') can continue, the appropriate goal and head terms must be isolated. In this implementation, a list of terms is represented by the pair

terms(lev, ts)

where

'lev' is the level of the activation record which introduced the clause that originated the terms and

'ts' is the list of terms as they appear in the program.

Get-goal-terms uses the second argument of the top-most activation record in state-in to obtain the goal terms.

In a similar way, Get-head-terms uses the third argument of the same activation record (in conjunction with the goal's predicate symbol and program), to isolate the head terms. It also produces a new state by replacing the constant 'UNKNOWN', which appears in the bindings position of the input state's most recent activation record, by an array (fixed length list) of entries with one entry for each new variable, each entry being the constant 'UNBOUND'.

The detailed specifications of Get-goal-terms and Get-head-terms are not given.

Match'

Matching proceeds incrementally, term-by-term, as follows:-

```
Match'(terms(lev-a, NIL), terms(lev-b, NIL), state, state)

Match'(terms(lev-a, t-a.ts-a), terms(lev-b, t-b.ts-b), state-in,
                                                        state-out)
  <-Static-Dynamic(lev-a, t-a, term-a) &
    Static-Dynamic(lev-b, t-b, term-b) &
    Unify(term-a, term-b, state-in, state-inter) &
    Match'(terms(lev-a, ts-a), terms(lev-b, ts-b), state-inter,
                                                   state-out)
```

Static-Dynamic

'Static-Dynamic' associates together the supplied level and static term to produce the corresponding term (we choose, for the sake of simplicity, to treat constants as 0-ary functors).

```
Static-Dynamic(level, t, var(level, t) )
  <- Variable(t)

Static-Dynamic(level, t, fun(level, t) )
  <- Functor(t)
```

Variable and Functor

A static variable is represented in the meta-language by the functor v(k) where k is a positive integer. A static functor is represented by the meta-level functor f(s, ts) where

´s´ represents the function symbol and

´ts´ represents the terms of the functor.

Thus the definitions of Variable and Functor are given by:-

```
Variable(v(k))
```

and

```
Functor(f(s, ts)).
```

Unify

In a structure-sharing implementation, any variable supplied to the unification process may or may not be bound. If it is bound, unification uses the term to which the variable is bound in place of the variable itself. Since a term may be a variable, this evaluation, or de-referencing, is in general recursive and we choose to separate it from the unification process proper (defined by Unify´).

Evaluation of both terms to be unified has the effect of avoiding variable-variable chains of bindings whenever possible. Thus if the variables u and v are two terms to be unified and u is unbound but v is bound to t then evaluation of both terms results in the binding u/t rather than u/v being made. Subsequent evaluation of u is thereby made more efficient. (Note, however, that chains of variable-variable

bindings may still arise. For example, if u and v (both initially unbound) are to be bound, the binding u/v (say) will be made. A later binding v/v' will mean that a subsequent evaluation of u will involve a non-trivial variable-variable binding chain.)

```
Unify(term-a, term-b, state-in, state-out)
  <- Evaluate(term-a, term-a', state-in) &
     Evaluate(term-b, term-b', state-in) &
     Unify'(term-a', term-b', state-in, state-out)
```

Evaluate

'Evaluate' follows the chain of variable-variable bindings until it comes across a variable bound to a functor or an unbound variable:-

```
Evaluate(term, term, state)
  <- Functor(term)

Evaluate(term-in, term-out, state)
  <- Variable(term-in) &
     Evaluate'(term-in, term-out, state)
```

Evaluate'

```
Evaluate'(var, term, state)
  <- Bound(var, term', state) &
     Evaluate(term', term, state)

Evaluate'(var, var, state)
  <- ¬Bound(var, term, state)
```

We defer the specification of 'Bound' for the moment.

Unify'

There are four self-explanatory cases to consider in the specification of Unify':-

```
Unify´(fun-a, fun-b, state-in, state-out)
  <- Functor(fun-a) &
     Functor(fun-b) &
     Unify-functors(fun-a, fun-b, state-in, state-out)

Unify´(var-a, var-b, state-in, state-out)
  <- Variable(var-a) &
     Variable(var-b) &
     Bind(var-a, var-b, state-in, state-out)

Unify´(var, fun, state-in, state-out)
  <- Variable(var) &
     Functor(fun) &
     Bind(var, fun, state-in, state-out)

Unify´(fun, var, state-in, state-out)
  <- Functor(fun) &
     Variable(var) &
     Bind(var, fun, state-in, state-out)
```

Unify-functors

If both terms are functors, the function symbols must be identical and the terms of the functions must be matched successfully:-

```
Unify-functors(fun-a, fun-b, state-in, state-out)
  <-Static-Dynamic( lev-a, f(s, ts-a), fun-a)&
    Static-Dynamic( lev-b, f(s, ts-b), fun-b)&
    Match´(terms(lev-a, ts-a), terms(lev-b, ts-b), state-in,
                                                   state-out)
```

The two ´Static-Dynamic´ calls are used "in reverse" here to produce static functors from dynamic ones.

Bind

We now turn to the question of how bindings are made and stored.

As was mentioned in Chapter 3, bindings are associated with the activation record at which the variable is introduced. If this activation record is not the most recent one (i.e. the one related to the current unification), an entry must be added to the reset list belonging to the most recent activation record.

```
Bind(var, term, state-in, state-out)
  <- Find-ar-and-add-binding(var, term, state-in, state-inter) &
     Parts(state-inter, ar-inter, state-rest)&
     Check-reset(var, ar-inter, ar-out) &
     Parts(state-out, ar-out, state-rest)
```

'Parts' holds if the first argument is a state whose most recent activation record is given by the second argument and the third argument represents the state of remaining activation records. Its first use here splits the supplied state into the top-most activation record and the state consisting of the remaining activation records; its second use is in the reverse direction - i.e. as a data constructor.

Find-ar-and-add-binding

```
Find-ar-and-add-binding(var, term, ar-in.state, ar-out.state)
  <- Local-var(var, ar-in) &
     Add-binding(var, term, ar-in, ar-out)

Find-ar-and-add-binding(var, term, ar.state-in, ar.state-out)
  <- ¬Local-var(var, ar) &
      Find-ar-and-add-binding(var, term, state-in, state-out)
```

The chosen representation of state means that a recursive search for the appropriate activation record is necessary. In a practical implementation, such access would be made through a direct reference, as previously explained. We could have simulated such direct access by representing the state as a relation of assertions, each assertion being, in essence, an activation record. We chose not to do this because such a description would have involved the deletion and insertion of clauses (to reflect destructive assignment in the ar's).

Add-binding

To avoid unnecessary detail, Add-binding is defined informally.

If the variable to be bound is var(lev, v(k)), Add-binding locates the k'th item in the fourth argument (bindings) of the input activation record (whose level is known to be lev). It produces an output activation record which differs from that input only insofar as the supplied term appears in place of the constant 'UNBOUND'.

Check-reset

Check-reset adds a reset entry to the list constituting the fifth argument of the input activation record if the variable in the first term is not local to that activation record. Otherwise, the output activation record is the same as that input.

```
Check-reset(var, ar, ar)
  <- Local-var(var, ar)

Check-reset(var, ar-in, ar-out)
  <- ¬Local-var(var, ar-in) &
     Add-reset-entry(var, ar-in, ar-out)
```

The lower-level procedure definitions are trivial and are not given.

Bound

Now that the method of adding bindings to the state has been described, it is easy to understand how to determine the term to which a given variable is bound.

The Bound relation holds between a variable and a term in the given state if the variable is bound to the term in that state. The Bound

relation does not hold if the constant 'UNBOUND' appears in place of a term.

The two steps involved in determining whether 'Bound' holds are

To find the activation record appropriate to the variable and

To succeed provided that the entry relating to the variable is not 'UNBOUND'.

Our implementation would simulate such direct accesses by two recursive searches, although we do not give the formal definitions here.

Again, such direct accesses could be more accurately simulated by representing bindings as assertions in a relation rather than as terms in a data structure.

Un-match

Now that matching has been specified, the converse, 'Un-match', called in the process of backtracking, is easily followed.

Referring to the context in which the 'Un-match' call is made, the steps required of the definition are

To once more make the fourth argument (bindings) of the most recent activation record equal to the constant UNKNOWN and

To use the reset list in order to undo all bindings for variables in that list. These bindings are each replaced by the constant 'UNBOUND'. The reset list is then made empty once more.

The formal specification is not given here.